NAVAL
POSTGRADUATE
SCHOOL

MONTEREY, CALIFORNIA

THESIS

USING ACTIVE NETWORKING TO DETECT AND TROUBLESHOOT ISSUES IN TACTICAL DATA NETWORKS

by

Kevin McMullen

June 2014

Thesis Co-Advisors:
Dennis Volpano
Raymond Buettner

Approved for public release; distribution is unlimited

THIS PAGE INTENTIONALLY LEFT BLANK

REPORT DOCUMENTATION PAGE		Form Approved OMB No. 0704–0188

1. AGENCY USE ONLY *(Leave Blank)*	2. REPORT DATE 06-20-2014	3. REPORT TYPE AND DATES COVERED Master's Thesis MM-DD-YYYY to MM-DD-YYYY	
4. TITLE AND SUBTITLE USING ACTIVE NETWORKING TO DETECT AND TROUBLESHOOT ISSUES IN TACTICAL DATA NETWORKS			**5. FUNDING NUMBERS**
6. AUTHOR(S) Kevin McMullen			
7. PERFORMING ORGANIZATION NAME(S) AND ADDRESS(ES) Naval Postgraduate School Monterey, CA 93943			**8. PERFORMING ORGANIZATION REPORT NUMBER**
9. SPONSORING / MONITORING AGENCY NAME(S) AND ADDRESS(ES) N/A			**10. SPONSORING / MONITORING AGENCY REPORT NUMBER**

11. SUPPLEMENTARY NOTES

The views expressed in this document are those of the author and do not reflect the official policy or position of the Department of Defense or the U.S. Government. IRB Protocol Number: N/A.

12a. DISTRIBUTION / AVAILABILITY STATEMENT Approved for public release; distribution is unlimited	12b. DISTRIBUTION CODE

13. ABSTRACT *(maximum 200 words)*

Troubleshooting in tactical data networks is often performed with a common toolset of programs, such as ping, traceroute, and protocols such as Simple Network Management Protocol. The assumption with such tools and protocols is that the logical configuration of the network is correct; if it is not, these tools could fail or return inconclusive results. While failure can be useful to prove a problem exists, it often does not provide enough data to actually diagnose the issue. Protocols such as Link Layer Discovery Protocol exist to troubleshoot from the data-link layer, but these protocols cannot operate between subnets. This limits their usefulness in tactical networks. An active networking project known as XPLANE has been developed at the Naval Postgraduate School with these issues in mind. XPLANE allows network operators to take active measurements in a network without relying on the logical layer. This ability is extremely important in live tactical networks, particularly when there is significant geographic separation between nodes. Before XPLANE can be used in tactical networks, important issues around security and the XPLANE's user interface must be resolved. This thesis explores the relevance of XPLANE in tactical networks and develops a front-end to XPLANE for tactical network operators.

14. SUBJECT TERMS Active Networking, Programmable Networks, Network Discovery			15. NUMBER OF PAGES 91
			16. PRICE CODE
17. SECURITY CLASSIFICATION OF REPORT Unclassified	18. SECURITY CLASSIFICATION OF THIS PAGE Unclassified	19. SECURITY CLASSIFICATION OF ABSTRACT Unclassified	20. LIMITATION OF ABSTRACT UU

Standard Form 298 (Rev. 2–89)
Prescribed by ANSI Std. 239–18

i

THIS PAGE INTENTIONALLY LEFT BLANK

Approved for public release; distribution is unlimited

USING ACTIVE NETWORKING TO DETECT AND TROUBLESHOOT ISSUES IN TACTICAL DATA NETWORKS

Kevin McMullen
Captain, United States Marine Corps
B.S., Penn State University, 2008

Submitted in partial fulfillment of the
requirements for the degrees of

MASTER OF SCIENCE IN COMPUTER SCIENCE

and

MASTER OF SCIENCE IN SYSTEMS TECHNOLOGY (COMMAND, CONTROL, AND COMMUNICATIONS)

from the

NAVAL POSTGRADUATE SCHOOL
June 2014

Author: Kevin McMullen

Approved by: Dennis Volpano
                 Thesis Co-Advisor

                 Raymond Buettner
                 Thesis Co-Advisor

                 Peter Denning
                 Chair, Department of Computer Science

                 Dan Boger
                 Chair, Department of Information Sciences

THIS PAGE INTENTIONALLY LEFT BLANK

ABSTRACT

Troubleshooting in tactical data networks is often performed with a common toolset of programs, such as ping, traceroute, and protocols such as Simple Network Management Protocol. The assumption with such tools and protocols is that the logical configuration of the network is correct; if it is not, these tools could fail or return inconclusive results. While failure can be useful to prove a problem exists, it often does not provide enough data to actually diagnose the issue. Protocols such as Link Layer Discovery Protocol exist to troubleshoot from the data-link layer, but these protocols cannot operate between subnets. This limits their usefulness in tactical networks. An active networking project known as XPLANE has been developed at the Naval Postgraduate School with these issues in mind. XPLANE allows network operators to take active measurements in a network without relying on the logical layer. This ability is extremely important in live tactical networks, particularly when there is significant geographic separation between nodes. Before XPLANE can be used in tactical networks, important issues around security and the XPLANE's user interface must be resolved. This thesis explores the relevance of XPLANE in tactical networks and develops a front-end to XPLANE for tactical network operators.

THIS PAGE INTENTIONALLY LEFT BLANK

Table of Contents

List of Figures

List of Tables

THIS PAGE INTENTIONALLY LEFT BLANK

List of Acronyms and Abbreviations

ACL	access control list
ARP	Address Resolution Protocol
CDP	Cisco Discovery Protocol
CORE	Common Open Research Emulator
CPS	continuation passing style
CSS	Cascading Style Sheets
D3	Data-Driven Documents
DOM	Document Object Model
EKMS	Electronic Key Management System
FDDI	Fiber Distributed Data Interface
FIPS	Federal Information Processing Standard
FOB	forward operating base
HTML5	Hypertext Markup Language Version 5
HTTP	Hypertext Transfer Protocol
ICMP	Internet Control Message Protocol
IED	improvised explosive device

IEEE	Institute of Electrical and Electronics Engineers
IETF	Internet Engineering Task Force
IP	Internet Protocol
JSON	JavaScript Object Notation
LLDP	Link Layer Discovery Protocol
LLTD	Link Layer Topology Discovery
MAC	media access control
MCCLL	Marine Corps Center for Lessons Learned
MIB	management information base
NRL	Naval Research Laboratory
NPS	Naval Postgraduate School
OEF	Operation Enduring Freedom
OSI	open systems interconnection
PLAN	Programming Language for Active Networks
PPP	Point-to-Point Protocol
RCT	regimental combat team
SDN	software defined networking
SIPRnet	Secret Internet Protocol Router Network
SSH	secure shell

SVG	Scalable Vector Graphics
SNMP	Simple Network Management Protocol
TCP	Transmission Control Protocol
TTL	time to live
UI	user interface
USMC	United States Marine Corps
XPL	XPLANE Programming Language

THIS PAGE INTENTIONALLY LEFT BLANK

Executive Summary

The author of this thesis deployed to Operation Enduring Freedom-Afghanistan in 2011-2012 as the communications officer for 2nd Battalion 11th Marines (2/11). During his time in Afghanistan, the author gained first-hand experience of managing and troubleshooting a network geographically dispersed throughout a combat zone. Configuration issues at remote sites affected network performance, which had impacts on 2/11's ability to do its mission. The majority of network issues were due to human error. These issues often rendered the Internet Protocol (IP) layer of the network, often called the logical layer, unreliable. The author quickly learned that common network troubleshooting tools such as ping, traceroute, and Simple Network Management Protocol have limited utility in logically broken networks. Basic troubleshooting became impossible, and Marines were often sent to remote sites to physically troubleshoot the network, which wasted valuable time and manpower.

The common suite of troubleshooting tools and protocols all suffer from the same weakness. They rely on the very thing that is often the source of the problem: the network's logical configuration. When a network has logical routing loops, unreliable transmission paths, and other anomalies, traditional tools become unreliable. Protocols such as Link Layer Discovery Protocol that operate from the data-link layer are bounded to their local subnet; they are unable to move between networks and are of limited use. An active networking system known as XPLANE has been developed at the Naval Postgraduate School with these issues in mind.

XPLANE utilizes the concept of active networking. The goal of the active networking paradigm is to increase capabilities in traditional networks by making them programmable. XPLANE programs run inside frames in the network. This allows XPLANE programs to discover and traverse network architecture to diagnose malfunctioning segments. XPLANE lives at the data-link layer of the network, which allows it to function in networks that are logically broken. It has unique capabilities that make it well suited for tactical networks. First, XPLANE programs can relocate throughout the network via the data-link layer. Second, programs are able to inject and capture packets in the network. When combined, these capabilities allow for active measurements to be taken from anywhere in the net-

work that has data-link layer connectivity. This feature could have a profound impact on troubleshooting in geographically dispersed tactical networks.

The problem with XPLANE is one of accessibility to the common network operator. In its current state, XPLANE consists of the runtime environment and a functional programming language to create XPLANE applications. There is no user interface or mechanism for tracking the results of multiple XPLANE programs. This thesis addresses these issues and creates a practical troubleshooting toolset for tactical networks. First, a comprehensive set of XPLANE applications known as the *tactical edge suite* has been developed. This suite of applications can map network topology from scratch and diagnose common network anomalies. Second, an *XPLANE server* has been developed that is responsible for launching XPLANE applications on the network and interpreting their results. The XPLANE server additionally keeps state of the network as discovered by the tactical edge suite. Finally, a *web-based user interface (UI)* for the XPLANE server has been developed. The web-based UI gives network operators a visual representation of the network and highlights any detected anomalies.

To test the XPLANE-based toolset, a small test network was created. This test network consisted of four routers running the XPLANE software and six client computers which did not run XPLANE. A number of anomalies were introduced into the network: duplicate and incorrect IP addresses were used, subnet addresses were entered incorrectly into routers, and logical routing loops were introduced. Network discovery and follow-on troubleshooting was first attempted with common troubleshooting tools discussed earlier. Attempts were then made to diagnose the same problems using the XPLANE-based toolset, and the results from both were compared. The XPLANE-based toolset outperformed traditional tools for both network discovery and identifying anomalies.

While this thesis addresses some of the issues that need attention before XPLANE becomes a practical tool, there remain ways it can still be improved. Some of them are discussed. They include security, improved packet injection, a more powerful user interface and a richer tactical edge suite of applications.

Acknowledgements

First, thanks to everyone that helped me through the process of writing this thesis. Most notably, thanks to Dr. Volpano for your guidance. You left no stone unturned during editing. Thanks for your patience.

Thanks to my wife and son for their support during this process. Sorry for all the late nights and busy weekends. I'll make it up to you.

Finally, thanks to all the Marines I have served with. I will never forget our experiences together. Thanks to Gunny Luke, Master Guns McCullough, and all the Staff NCOs that set me straight. Thanks to the Data Marines who put up with me over the years to include Barker, Chamberain, Wilson, Hagerman, Cooley, and Hughes. There was never a dull moment. Semper Fidelis.

THIS PAGE INTENTIONALLY LEFT BLANK

CHAPTER 1:
Introduction

In late 2011, the author of this thesis deployed to Operation Enduring Freedom (OEF) as the communications officer for 2nd Battalion 11th Marines (2/11). As the only artillery battalion in Helmand Province, 2/11's mission was to provide supporting fires to any unit in the area of operations, which stretched from the Kajaki Dam to the Helmand/Pakistan border. In order to cover such a large area, 2/11 was segmented into small, platoon-sized elements that were geographically dispersed throughout Helmand Province to provide maximum coverage. As a consequence of this dispersion, 2/11 became extremely reliant on Helmand's Secret Internet Protocol Router Network (SIPRnet) in order to receive and process calls for fire. Most units were well out of radio range from the artillery position that supported them, and the protocols to have fire missions approved by higher headquarters often required the use of SIPRnet for coordination.

By the midpoint of his deployment, the author had hit a wall. Nearly every artillery position in Helmand Province was having serious SIPRnet connectivity issues, which was causing delays in fire missions. Delays in fire missions meant that Marines in a firefight on the ground would have to sit there and wait for the support they requested, which was dangerous and unacceptable. The problem became so bad that meetings were occurring on a day-to-day basis between the 2/11 Communications section and the Division G-6, which was the agency responsible for the entire Helmand network. The conclusion of most troubleshooting sessions was the same; the problem was often too difficult to diagnose and fix from the central hub of the network, Camp Leatherneck, and someone would have to travel to different forward operating bases (FOBs) and physically troubleshoot the issue. By the end of the deployment, the author and his Marines had travelled to five different FOBs throughout Helmand Province to physically troubleshoot issues. These issues almost always ended up being caused by human error. Other units to include the Division G-6 and the regimental combat team (RCT) 8 and RCT 6 Communications Section also travelled to artillery firing positions to troubleshoot the network throughout 2/11's deployment.

Sending Marines to fix network issues was not a trivial decision. The threat of improvised

explosive devices (IEDs) prevented most personnel from travelling via a ground convoy, especially for long distances. This made aircraft the primary mode of personnel transportation across Afghanistan. Unfortunately, travelling by aircraft was also risky, and at best would take two to four days for a round trip flight, depending on weather. All of these issues caused the author to re-evaluate the situation. How did it get this bad? How did sending Marines become the fix to every network issue we experience? The answer was simple: the network was too dispersed and too complex for the tools available at the time. The Helmand communications network had become a mesh of interconnected FOBs spanning across hundreds of kilometers. The redundant architecture, while tactically sound, made equipment configurations nearly impossible to troubleshoot from a central location. Simple tools like ping and traceroute were unable to help, mostly due to the fact that such tools rely on the very thing that needs to be fixed: the network configuration. More complex tools such as Solar Winds were in use, but unfortunately could only offer a picture of what they could see from the central hub of the network, Camp Leatherneck, which added no value. The last bastion of hope, the documentation for the entire Helmand Province network, was only accurate up to late 2011. What the Marines of Helmand Province needed were tools that would allow them to ask deeper questions about the network. They needed tools that could utilize the network segments that were working correctly, such as the link-layer of the various transmissions systems, to gain insight into the malfunctioning upper layers. Such tools could have prevented the need to send additional Marines into harm's way, simply to troubleshoot network segments that were physically out of reach.

1.1   Overview and Problem Statement

Network troubleshooting is often performed with a suite of commonly available tools such as ping, traceroute, and Simple Network Management Protocol (SNMP). These tools can be used to perform common tests such as connectivity between nodes, path discovery between nodes, and in the case of SNMP, statistics and detailed information on node configurations can be queried from a distance. The assumption with all such tools is that the logical configuration of the network is correct. If it is not, tests performed with these tools will likely fail. While failure can be useful to prove a problem exists, it often does not provide enough data to figure out why the problem exists in the first place.

The common weakness in these tools lies at the heart of the original assumption made:

2

how can tools that rely on the logical layer diagnose issues with the logical layer itself? The obvious answer is they often cannot. In order to troubleshoot issues in the logical layer, one has to move "back down" the layers of the network. In the case of the open systems interconnection (OSI) model, this means having tools that work at layer two, often called the data link layer. At the data link layer, there are a number of protocols that aid in network discovery such as the Address Resolution Protocol (ARP) [1], Link Layer Discovery Protocol (LLDP) [2], and Microsoft's Link Layer Topology Discovery (LLTD) [3]. There are also tools such as linkloop which replicate the behavior of the ping utility [4]. Unfortunately, all of the aforementioned tools and protocols only aid in one aspect of network troubleshooting: discovery. Additionally, such tests are limited to the local broadcast domain, which means a node can only discover what it is directly connected to. This is a limitation of operating at the data link layer; the network is organized into non-routable segments that are unable to communicate without relying on layer three, the logical layer.

1.2 Research Questions

The need for tools that operate at the data link layer, yet are able to traverse networks beyond the local broadcast domain is apparent. Such tools could take advantage of physical connections that still exist between nodes to troubleshoot logical layer issues. What would the requirements of such a tool be? With no routing information inherently available at the data link layer, packets would have to somehow be able to make decisions about their path as they traverse the network. In other words, packets would need to be active instead of a collection of passive data. This concept, known as active networking, is not a new one [5]. The software defined networking (SDN) paradigm, which has gained popularity in recent years, has its roots in the idea of programmable networks [6]. By extending the functionality of traditional networks through active networking, new systems can be developed to manage, troubleshoot, and even reconfigure a running network in ways not possible today.

One such system, known as XPLANE, is currently in development at the Naval Postgraduate School (NPS) [7]. Once enabled on a network, XPLANE allows for small programs to be run on the network itself. XPLANE programs have two key capabilities: the ability to crawl through the network via the data link layer, as well as the ability to inject

3

and capture packets in the network [7]. Used together, these capabilities allow for active measurements to be performed from any point in the network that has link-layer connectivity. When considered in the context of tactical data networks, capabilities such as these have the potential to change the way network troubleshooting is performed. This thesis will explore the following question: how can active networking systems such as XPLANE improve troubleshooting in malfunctioning or damaged tactical data networks?

The XPLANE has been demonstrated in laboratory settings, however work remains to make it a practical tool for troubleshooting tactical networks. First, a set of XPLANE applications that would aid in troubleshooting common issues in tactical networks needs to be identified and developed. Second, an interface needs to be developed for launching these applications and capturing the results. XPLANE applications run asynchronously on the network; results of any measurements taken are returned back to the node that initiated the program and then forgotten. It is up to the user to keep state and interpret results in a meaningful way. In order to provide accessibility to the XPLANE results, a visual user interface needs to be developed. Finally, introducing new software such as XPLANE into a network presents security risks. What are the security risks, and how will they be mitigated?

To address these issues, a comprehensive set of XPLANE applications known as the *tactical edge suite* has been developed to map the topology of a network and diagnose common network anomalies. These applications are managed and launched via a *XPLANE server* that is responsible for capturing and interpreting results of programs ran on the network. The XPLANE server is also responsible for keeping state of the network, which can be visualized via a *web-based user interface (UI)* to the server. The XPLANE web-based interface displays the state of the network as a graph of interconnected nodes. The interface will also highlight any anomalies detected in the network. Security concerns with XPLANE have been identified, and access control mechanisms have been incorporated to ensure that only authorized users are allowed to execute code on the network. Additionally, the topic of key management has been explored and a model for access control has been developed.

1.3 Scope of Research

The research conducted in this thesis will be limited to detection and diagnosis of issues in tactical data networks; no attempt will be made to remotely fix issues diagnosed with XPLANE. Changing the configuration of live network devices presents its own set of challenges that include dealing with multiple proprietary interfaces and access control mechanisms [6]. Additionally, XPLANE is limited to observing network behavior by design and therefore is not capable of modifying network behavior. Possible extensions to this thesis and the results provided will be discussed in Chapter 7 under future work.

1.4 Thesis Organization

Chapter 2 begins with an overview of the OSI model and basic networking concepts. Common issues in tactical data networks are presented, and the traditional tools and protocols used in network troubleshooting are explained, with emphasis on their shortfalls in tactical environments. Recent advances in network management systems is presented, and the concept of active networking is explained. XPLANE, an active networking system, is introduced. Active networking research projects that inspired XPLANE's design are addressed, and their limitations in tactical environments are discussed.

Chapter 3 presents XPLANE and XPLANE Programming Language (XPL). An overview of XPLANE and XPL's capabilities and limitations is given. The mechanics of XPLANE at the data-link layer of the network is explained. Programming in XPL is addressed, and code examples are given to demonstrate XPL constructs to the reader.

Chapter 4 presents the tactical edge suite for troubleshooting tactical networks. The design philosophy for the suite is presented, along with the testing methodology and platforms used. For each application developed, the rationale behind developing it is explained, as well as a general description of the algorithm, any assumptions made, and additional concerns.

Chapter 5 presents the design of an XPLANE server, and the design of the web-based UI. The composition of the server and its integration with the tactical edge suite is presented. The web-based UI and its interface to the XPLANE server is described.

Chapter 6 presents the results of an assessment made of the utility of the XPLANE server

and the tactical edge suite. The XPLANE server and tactical edge suite are compared with traditional network tools for troubleshooting networks. The results of this comparison are given.

Chapter 7 provides some conclusions and addresses future work of both XPLANE itself as well as the work presented in this thesis.

Appendix A contains all code listings referenced in this thesis.

Appendix B contains the detailed configuration for the network used in Chapter 6.

CHAPTER 2:
Background and Related Work

Computer networks have greatly evolved over the past few decades. The amount of devices on networks, as well as the functionality of each device, has increased over time [6]. Traditionally, networks have been configured "by hand," (i.e., manually configured). As of this writing, this is the standard method of network configuration in the United States Marine Corps (USMC). Routers, switches, firewalls, and combination devices must be configured in a logical manner that allows for proper operation of all devices on the network [6]. As the amount of devices on a network increases, manually managing all devices and their configurations from a central point becomes difficult. In the case of geographically-dispersed networks in an active combat zone, the idea is untenable. In order to address the issues of managing complex networks, one has to understand why traditional tools and protocols have hit the limits of their capabilities, as well as previous and current work related to these issues.

2.1 OSI Model

The OSI model is a concept used to describe the various layers of computer networks and how the layers relate to each other [1]. The model breaks networks down in to seven distinct layers: physical, data link, network, transport, session, presentation, and application [1]. Layers are typically numbered from one to seven starting from the physical layer, and each layer is defined by the function it performs in the network [8]. This thesis will focus on the data link layer (layer two) and the network layer (layer three).

2.1.1 Data Link Layer

The function of the data link layer is to provide error-free transmission between adjacent nodes over the physical layer [8]. Individual data packets are referred to as *frames* at this layer [1]. Many different link layer protocols exist; common examples are Point-to-Point Protocol (PPP), Fiber Distributed Data Interface (FDDI), and the Institute of Electrical and Electronics Engineers (IEEE) 802.3 standard, more commonly referred to as *Ethernet*. Ethernet is the most common link layer protocol in use today [1]. As a service, Ethernet offers

flow control, error detection, and error correction on data transmitted between adjacent nodes [1]. This thesis will focus on using Ethernet as the link layer protocol.

2.1.2 Network Layer

Transmission of data at the link layer is limited to adjacent nodes. In order to route packets between networks, the link layer forwards the contents of frames to the network layer. This layer is also commonly referred to as the *logical layer*. The function of the network layer is to provide end-to-end delivery of data between nodes of different networks [1]. The most common network layer protocol is the IP. IP provides end-to-end delivery of *datagrams*, i.e. packets, via best effort delivery [1]. This thesis will focus on using IP (version 4) as the network layer protocol.

2.2 Network Issues that Require Troubleshooting

The logical layer of the network, in particular IP, is necessary to create larger networks from subnetworks [1]. The ability to route packets from one network to another is what makes networks such as the Internet possible. In order to route packets efficiently, IP relies on *routing algorithms* to calculate paths on its behalf [1]. Routing algorithms in turn rely on a set of assumptions about the network and how it is configured. One important assumption is that each node in the network has a unique *IP address*, which identifies it within the network. Another assumption is that the routers themselves have been correctly configured. When these assumptions are proved false, errors will begin to occur in the network. This thesis will refer to network errors as *anomalies*.

2.2.1 IP Address Anomalies

IP addresses are used to uniquely identify nodes within a network. When two or more nodes use the same IP address, this is referred to as *duplicate IP addresses* within the network. Duplicate IP addresses present a problem to network devices; how will a decision be made on where to forward data when there are multiple nodes using the same IP address? The behavior of network devices in this situation will depend on many factors, such as ARP table caches.

Along with IP addresses comes the notion of *subnets and subnet masks*, which are used to segment networks into distinct sub-networks [1]. Subnet masks have to be correctly

configured on nodes and routers, or else routing algorithms will be unable to make a determination on *which* subnetwork a given IP address lives. Subnet masks can be accidentally misconfigured on routers, end hosts, or both. The effect on the network will depend on many factors, making misconfigured subnet masks difficult to remotely diagnose and troubleshoot.

2.2.2   Packet Loss

IP is a best effort service, and therefore makes no guarantees that packets will be delivered to their destination [1]. For reliable end-to-end transmission, protocols such as the Transmission Control Protocol (TCP) are implemented at layer four of the network stack, the transport layer [1]. Packet loss can occur in a network for a variety of reasons; examples include transmission errors, routing loops, firewalls, access control list (ACL) restrictions, and queueing delays in routers. While it is easy to detect packet loss, it is often hard to single out the source of the error.

2.2.3   Routing Loops

Routing loops occur in networks when packets are unable to leave an infinitely looping path in the network [9]. Figure 2.1 shows an example of a routing loop that has been formed between three routers in a network. Routing loops can occur for many different reasons; examples include incorrectly configured static routes, delays in network convergence, and general router misconfiguration [9]. As a network anomaly, routing loops have unique side effects depending on the network topology and configuration, and are therefore hard to detect in many networks [9].

2.3   Traditional Tools for Troubleshooting

Over time, network administrators have come to rely on a common suite of tools and protocols to troubleshoot networks. These tools are usually available on all end-user operating systems as well as network devices such as routers. This section will discuss the more common tools and protocols to address the shortfalls of each one. This section is not a comprehensive list of every troubleshooting tool in existence; there are simply too many. Most if not all are similar enough in function to the tools and protocols that are addressed, and suffer from the same weaknesses.
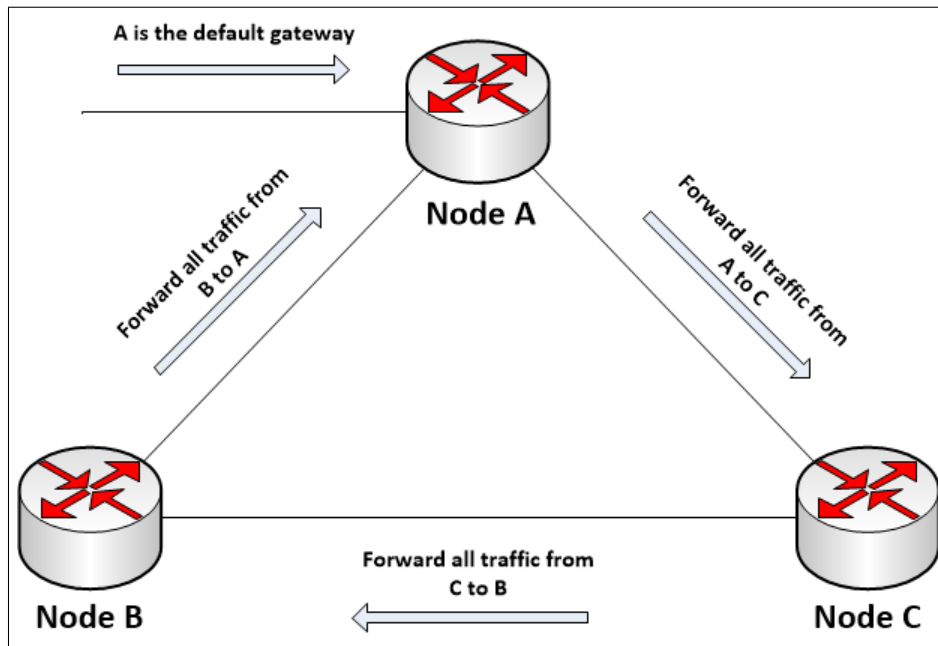
Figure 2.1: An example of a routing loop between three nodes.

2.3.1 Ping

The *ping* utility is a common troubleshooting tool that uses the Internet Control Message Protocol (ICMP) to test end-to-end connectivity between nodes. It typically comes in the form of a command-line tool, however the basic ICMP *echo request/response* mechanism is often referred to as "pinging a host", regardless of the way the message was sent. A successful ping implies end-to-end connectivity between two nodes on a network. A failed ping does not necessarily imply a lack of connectivity between two nodes; it simply indicates there *could* be an issue. Host-based firewalls, network firewalls, ACL restrictions, and other factors could all cause ping to fail, even if there is a logical path between the two nodes in question. This limits ping's utility in troubleshooting environments.

2.3.2 Traceroute

The *traceroute* utility is another command-line troubleshooting tool. Traceroute makes use of the time to live (TTL) field of IP datagrams to trace the route taken to a given destination [1]. A traceroute analysis of the path between two nodes will yield a list of hops along the path, the round-trip delay to each hop, and the hostnames and/or IP addresses of each

node along the path [1]. Just like ping, traceroute utilizes ICMP to send messages. There are versions that can alternatively use TCP for networks that restrict ICMP messages. The information provided by traceroute can be very useful in certain troubleshooting situations. When the logical layer of the network is malfunctioning, the utility of traceroute can vary widely.

2.3.3   SNMP

SNMP is a network management protocol used to manage devices and query information from them over a network [1]. SNMP organizes information into objects known as management information bases (MIBs). MIBs are used to organize information in a hierarchical fashion with a well-defined structure that is standard across all devices. In turn, MIBs can be viewed as a database of configuration parameters and statistics that can be reached from other nodes in the network [1]. There are standard MIBs for information such as device configuration, protocol statistics, and routing tables and configurations.

In comparison to command-line utilities such as ping and traceroute, SNMP is an actual management protocol with much more capability. SNMP has far more utility in certain troubleshooting situations than ping and traceroute, but unfortunately suffers from the same weaknesses. SNMP is a network layer (layer three) protocol, and its capability in a malfunctioning network varies widely.

2.3.4   LLDP, LLTD, and CDP

The IEEE LLDP, Microsoft's LLTD, and Cisco's Cisco Discovery Protocol (CDP) are link layer protocols used by nodes to advertise information to the rest of the local area network [2, 3, 10]. This information can include information such as a node's identity, neighbors, and services available [2, 3, 10]. The three protocols are similar enough in functionality to address their capabilities and limitations as a group. While LLDP, CDP, and LLTD are useful as link layer discovery tools, they are by definition limited to the local broadcast domain and therefore unable to help in discovering nodes across network boundaries.

2.4   Recent Advances in Network Troubleshooting

In addition to the tools and protocols mentioned, there are a number of successful academic projects that have created new and more powerful network management and troubleshoot-

ing capabilities.

2.4.1 Sophia

Sophia is a project aimed at developing a network *information plane* to collect, store, propagate, and react to observations about the network in a distributed manner [11]. Sophia is a good example of the *overlay network* concept, which is a networked system of nodes within the larger network. Sophia creates an overlay network of distributed sensors that provide information to a declarative programming environment that evaluates logic statements about the network [11]. Sophia is novel in the fact that administrators can query the network directly for management information in a declarative manner [11]. In other words, administrators can ask deep questions about network state, without telling the network *how* to find the answer to the query. While Sophia shares similar goals with XPLANE, it relies on the assumption that the logical layer of the network is functioning correctly, limiting its usefulness in malfunctioning networks.

2.4.2 NetQuery

NetQuery is a project from Cornell University aimed at implementing a *knowledge plane* on large-scale networks such as the Internet [12]. Similar to Sophia, NetQuery aims to disseminate information about network entities to support application-level reasoning. NetQuery focuses on trusted computing, information attribution, and how to reason about the network with limited trusted information [12]. NetQuery, like Sophia, makes assumptions on the correctness of the logical layer of the network, limiting its usefulness in malfunctioning networks.

2.5 Active Networking

The term *active networking* refers to a new networking paradigm developed in the 1990s [5]. The goal of this paradigm was to generate innovation in the design and control of networks by making them programmable [6]. By making networks programmable, followers of the active networking paradigm were looking for novel ways to create new capabilities and services. Traditional computer networks are not programmable in the classic sense; devices are individually configured, and there is no environment on the network for code to be executed within [6]. Active networking changes this by presenting two candidate network programming models; this thesis will focus on the *capsule model* [6]. In the capsule

model, packets are a combination of code and data known as *capsules* [5]. The code in these capsules can be executed at any node in the network and can modify network behavior [5].

This radical approach to opening up network capabilities may have been ahead of its time; active networking never experienced widespread adoption beyond the research community [6]. A possible reason for this was the lack of a clear demand for such capabilities at the time [6]. This is no longer the case. In many ways, the active networking community was attempting to address issues now being addressed by SDN [6]. While there are similarities between SDN and active networking, SDN is primarily concerned with the idea of separating the *control plane* of a network from the *data plane* in order to dynamically adjust the control plane based on network conditions [6]. In other words, SDN intends to make networks more dynamic by standardizing the control of network devices such as routers and switches from a central location. Active networking in the capsule model focuses on manipulation from the data plane instead; this means that it is the responsibility of the capsules, not the routers, to dynamically adjust to network conditions [6]. The importance of this distinction in tactical networks will be explained further in Chapter 3. SDN and the centralization of the control plane will likely have a role in future tactical networks, but it will not be a focus of this thesis.

The active networking paradigm has manifested itself through a number of academic projects [6]. XPLANE is only one of them. In order to understand why XPLANE is a better fit for tactical networks than the others, we need to address prior and related work.

2.5.1   PLAN

Programming Language for Active Networks (PLAN) is an active networking project developed at the University of Pennsylvania in the late 1990s. The goal of PLAN was to develop a functional programming interface to the concept of active networking [13]. The developers of PLAN envisioned using it as a network-level 'glue' language for services written in other general-purpose languages [13]. Diagnostic programs such as ping and traceroute can be emulated in PLAN with minimal effort, and more importantly, without relying in ICMP. This kind of flexibility demonstrates the ability to replace the functionality of certain protocols with simple programming interfaces [13]. The design of XPLANE was partially inspired by PLAN, but the two platforms differ in an important aspect; PLAN

lives at the network layer, while XPLANE lives at the data link layer [7, 13]. Additionally, PLAN is not focused on network troubleshooting, and lacks features such as packet injection to take active network measurements. The PLAN project is no longer maintained.

2.5.2 Sprocket

The Sprocket programming language is a component of the *Smart Packets* architecture, an active networking project focused on network monitoring and management [14]. Similar to PLAN and XPLANE, Smart Packets are active network programs that execute on nodes as they traverse the network [14]. Sprocket, a programming language based on the syntax of C, serves as the high-level language that Smart Packets are written in [14]. The design of XPLANE and the XPL is influenced by Smart Packets and Sprocket [7]. As with PLAN, Smart Packets operates at the network layer and therefore has limited use in the types of networks this thesis intends to explore. Smart Packets is not focused on network troubleshooting, as is the case with PLAN, and similarly lacks active network measurement capability.

The Active Networking research efforts discussed so far have a common weakness: they are either unable to function in logically broken networks, or were not specifically designed to troubleshoot networks. Both of these shortfalls limit the usefulness of such systems in tactical environments. An active networking system known as XPLANE has been developed with these issues in mind. The next chapter will introduce XPLANE and give an overview, with code examples, of XPLANE's capabilities.

CHAPTER 3:
XPLANE

XPLANE is an active networking system developed at NPS. The goal of XPLANE is to develop a system that can perform active measurements for troubleshooting in a network relying solely on physical connectivity [7]. In other words, XPLANE aims to function correctly *even if the networking is logically broken*. This feature, along with programmable packet injection, make XPLANE suitable for troubleshooting and separate it from other active networking efforts. This thesis will focus on using XPLANE to diagnose anomalies in networks, and compare the utility of information provided by XPLANE against the common troubleshooting tools and protocols mentioned earlier.

3.1 Overview

XPLANE can be considered an overlay network that exists just above the data link layer [7]. The system consists of a *shim* that operates at the data link layer of network devices, and a programming language, XPL, that is used to create programs that will run on the XPLANE. The shim provides the runtime environment for XPLANE programs. When an XPLANE packet is received on an XPL-enabled node, the shim decodes the packet and executes the XPL code inside. Figure 3.1 shows the format of an XPLANE packet. The current version of XPLANE does not utilize the *Sender ID*, *Sequence Number*, or *Fragmentation* fields. It is important to note that the marshaled code inside the packet might be the continuation of an ongoing computation in the network. XPL programs can *relocate* themselves throughout the network during the execution of a program as they need [7].

Marker	Version	Packet Length
Sender ID		Receiver ID
Sequence Number		
Fragmentation		Checksum
Authentication (20 bytes)		
Marshaled Code		

Figure 3.1: XPLANE packet format

XPLANE does not rely on the logical layer for transportation through a network. It instead relies on *link layer broadcasts* between nodes. This allows XPL programs to move through networks that are logically broken even when normal IP traffic cannot. If an executing program chooses to relocate, the XPLANE shim will generate an XPLANE packet with the execution's current state stored in it, and broadcast the frame on the data link layer to all adjacent nodes. XPLANE uses the notion of XPLANE node IDs, which are unique node identifiers throughout the XPLANE. Nodes identify themselves as recipients of an XPLANE packet based on the node ID of the receiver. Packets can be addressed to specific node IDs. Packets can additionally be addressed to all adjacent nodes. XPLANE refers to this as *flooding*. XPLANE programs aim to fit in a single Ethernet frame, which has a maximum size of 1500 bytes. XPLANE will remember neighboring XPLANE nodes once they are discovered. The XPLANE shim can also be configured to *beacon* its presence on the local subnet to all other XPL-enabled nodes. Beaconing helps in discovering directly connected neighbors, increasing the utility of XPL programs that rely on neighbor information.

3.2 XPL

XPL is a subset of the TinyScheme functional programming language. It has built-in constructs to perform functions and access node information specifically related to XPLANE. Table 3.1 gives a summary of common XPL-Scheme functions, their pseudocode syntax for this thesis, and a description of functionality. The first feature of XPL to explore is the ability to relocate to a different node. This can be accomplished in two different ways with XPLANE. The first way is the *On* function, which has the syntax:

$$On(e1, e2)$$

where *e1* is an expression that evaluates to the node ID of a directly connected neighbor, and *e2* is the expression to evaluate on that neighbor. The expression evaluated on the neighboring node can be any legal expression, including more calls of the *On* function:

$$On(1, On(2, (On\ 3, (2 + 2))))$$

This program would relocate to the XPLANE node with node 1, then relocate to node 2, then relocate to node 3, and finally evaluate the expression **2+2** on node 3. Figure 3.2 shows how the computation would take place on a network assuming the nodes are directly connected.

XPL-Scheme	Pseudocode	Description
(on n e)	$On(n, e)$	Relocate to node n and evaluate expression e
(onflood e)	$OnFlood(e)$	Relocate to all directly-connected neighbors and evaluate expression e
node	$Node()$	Returns node ID for current node
(node.ifaces)	$Node.ifaces()$	Returns list of network interfaces for current node
(node.ip i)	$Node.ip(i)$	Returns IP address for interface i on current node
(node.mask i)	$Node.mask(i)$	Returns subnet mask for interface i on current node
(node.ethaddr i)	$Node.ethaddr(i)$	Returns MAC address for interface i on current node
(node.direct i)	$Node.direct(i)$	Returns list of known neighbors on interface i of current node
(send e1 e2 e3)	$Send(e1, e2; e3)$	e1 evaluates to device to send on, e2 evaluates to list specifying protocol and destination, and e3 represents a body of code to be evaluated
(pcap e1 e2 e3 e4)	$Pcap(e1, e2, e3, e4)$	e1 evaluates to device to listen on, e2 evaluates to protocol, e3 evaluates to timeout value for listener, e4 evaluates to packet processing function
(list ...)	$List()$	Returns arguments to function as a list
(car l)	$Head(l)$	Returns the first element of the list l
(cdr l)	$Tail(l)$	Returns all but the head of the list l
(append l ...)	$AppendToList(l, ...)$	Appends all supplied arguments after list l to l
(member s l)	$Member(s, l)$	Returns *true* if element s is a member of list l. Returns *false* otherwise

Table 3.1: XPL-Scheme and Pseudocode syntax for XPLANE functions

A more useful example of code relocation is to retrieve information about a node on the network:

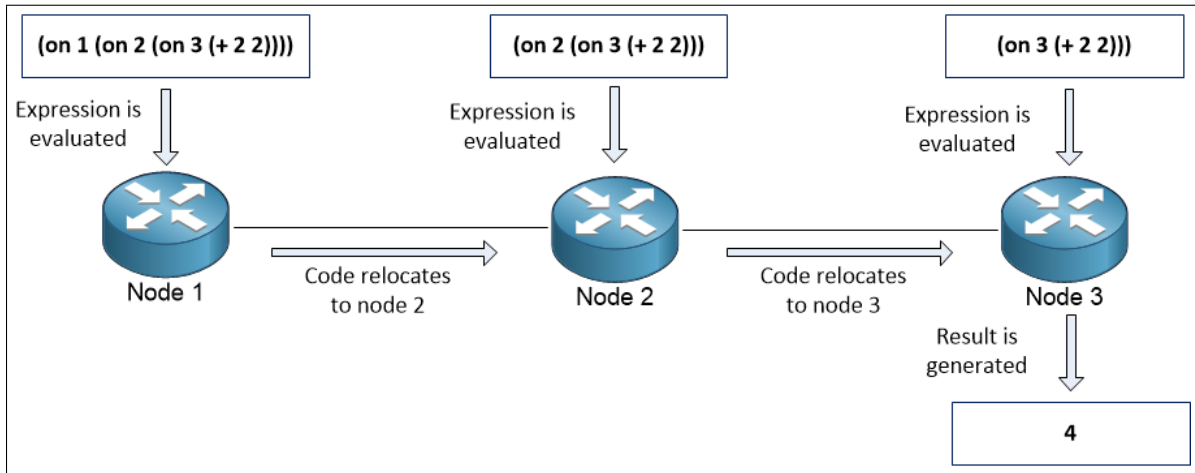$$On(1, On(2, On(3, (Node.ip(ai)))))$$

Figure 3.2: Visualization of relocation across multiple nodes in XPL

This program will traverse the network in a similar fashion, evaluating the expression *Node.ip(ai)* on node 3. *Node.ip()* is a function in XPL that takes a network interface as an argument and returns the IP address of that interface. *ai* is a variable in XPLANE representing the network interface that the XPLANE packet arrived on. The result of the expression *Node.ip(ai)* would then be the IP address of the interface the program arrived on. Unfortunately, the result of this computation would stay on node 3. A more useful program would somehow return the result of the program to the node of origin. This is accomplished in XPLANE through the use of continuation passing style (CPS) transformations.

3.2.1 Continuation Passing Style

CPS is a programming technique in computer science in which each function or procedure that is called is passed a *continuation*. This continuation represents the remaining work of the computation to be performed, and is executed at the end of function instead of returning to the caller. This allows a program to continue execution without ever having to return to the point from which it was called [15]. XPLANE utilizes CPS to accumulate a continuation that will return the result of a computation back to the originating node. When the result of a program is generated, the continuation is called, carrying the result back to the originating node along the reverse path taken to the final node.

In addition to the *On* function, XPL has a function, *OnFlood*, which has the syntax:

OnFlood(e1)

where *e1* is any valid expression. *OnFlood* will evaluate the expression *e1* on all directly connected neighbors, hence the name. Consider the following program:

OnFlood(Node.ip(ai))

This program would flood to all directly connected neighbors which would in turn evaluate *Node.ip(ai)* and return the IP address of the arrival interface. If this code were compiled with the CPS transformation, the IP address of every directly connected neighbor would be returned to the origin. Figure 3.3 shows how the computation would take place on a network assuming the originating node has three directly connected neighbors.
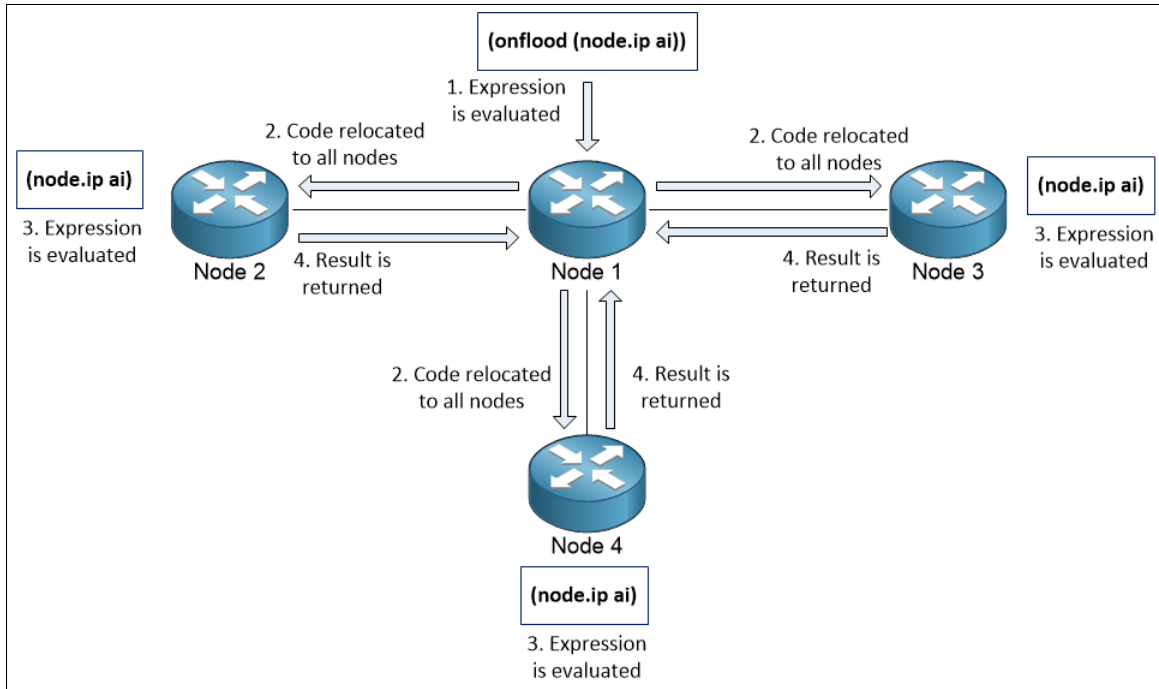
Figure 3.3: Visualization of program utilizing OnFlood compiled with CPS transformation

It is interesting to explore the mechanics of the above *OnFlood* program. By calling *OnFlood*, the current state of execution was suspended, encapsulated into an XPLANE packet, and broadcasted on the network. In effect, this made *n* copies of the original program, where *n* is the number of directly connected neighbors that are XPL-enabled. When used

19

in a recursive function, it is possible to utilize *OnFlood* to reach every XPL-enabled node in a network non-deterministically, assuming there is a subgraph connecting all such nodes. This is a powerful feature of XPL. Programs need not have any prior knowledge of network topology, and can make very few assumptions about it. A good example of a use-case for this feature would be locating an XPL node on the network without any prior knowledge of its location. Such a program would follow the general algorithm:

Algorithm 1 Algorithm to locate a node in the XPLANE

1: **procedure** FINDNODE($n, path$)   ▷ n=target node, path=list of nodes visited so far
2:     $currentNode \leftarrow Node()$
3:     **if** $currentNode = n$ **then**
4:         **return** $path$ ▷ path contains all nodes from source to destination
5:     **else if** $Member(currentNode, path) = false$ **then**
6:         $path \leftarrow AppendToList(path, currentNode)$ ▷ Append the current node to path
7:         $OnFlood(FindNode(n, path))$
8:     **end if**
9: **end procedure**

In this algorithm, the *Member()* function returns true if the first argument is a member of the second argument, and the *AppendToList()* function appends the first argument to the list provided as the second argument. An XPL implementation of this algorithm can be found as Listing 1 of Appendix A. The *FindNode()* procedure will return the path taken to the destination node if a path is found. It is worth noting that *all* paths to the destination node will be found due to the use of *OnFlood*, and therefore the originating node may receive multiple results.

3.2.2   Packet Injection and Capture

The second major feature of XPL is packet capture and injection. The XPL functions *Send* and *Pcap* are used to send and capture packets, respectively. The two are often used in conjunction to perform some kind of active measurement within the network. The syntax of *Send* is as follows:

Send(iface, pktdesc; body)

where *iface* is a network interface, *pktdesc* is a packet descriptor and *body* is the XPL program within scope of the sent packet. The packet descriptor is used to specify parameters of the packet to inject such as protocol and destination host. The *body* will be called after the packet is injected. This is typically used to schedule packet capture via the *Pcap* function. *Pcap* has the syntax:

$$Pcap(iface, filter, timeout, pktfunc)$$

where *iface* is a network interface, *filter* is a tcpdump-style filter, *timeout* is a timeout value, and *pktfunc* is a packet-processing function. The *filter* will instruct *Pcap* to only capture packets that match the supplied filter. The *timeout* value specifies the number of seconds to capture on the supplied interface. The packet-processing function is responsible for parsing captured packets and making sense of any captured traffic.

For example, Algorithm 2 demonstrates a ping-like program using *Send* and *Pcap*.

Algorithm 2 Send a ping and look for reply

1: **procedure** P($pkts$)   ▷ pkts=list of packets to process
2:     **if** $pkts = null$ **then**
3:         **return** $false$ ▷ reply packet was not found
4:     **else**
5:         $pkt \leftarrow Head(pkts)$ ▷ grab the first packet from the list
6:         **if** $IP.src(pkt) = $"10.0.0.1" **then**
7:             **return** $true$ ▷ reply packet was found
8:         **else**
9:             $P(Tail(pkts))$ ▷ continue processing remaining packets
10:         **end if**
11:     **end if**
12: **end procedure**
13: $protoAndDest \leftarrow List($"*EchoRequest*","10.0.0.1"$)$
14: $Send(1, protoAndDest; Pcap(1,$"*icmp*"$,3,P))$

The program attempts to ping the IP address 10.0.0.1 and looks for a reply. The program begins by calling *Send* to generate and inject an "EchoRequest" packet to the IP address 10.0.0.1 on the supplied interface, which in this case is interface 1. After the packet is injected, *Send* evaluates the supplied body which calls *Pcap*. *Pcap* schedules packet capture on the supplied interface (again interface 1) and filters for ICMP packets. *Pcap* will attempt to capture packets for three seconds before terminating. Any captured packets will be handled by the specified packet-processing function *P*. *P* takes a list of captured packets and looks for replies from 10.0.0.1, returning true if one is found and false otherwise. Listing 2 of Appendix A gives the XPL code for this algorithm.

By combining packet capture and injection with relocation, packets can be inserted into the network at one node, and captured at a different node. This kind of flexibility allows for flexible real time measurements in a network.

CHAPTER 4:
Developing the Tactical Edge Suite of Applications

This chapter will begin to explain the rationale behind the design and development of a new troubleshooting toolset. The core of this toolset will consist of a library of XPLANE applications that specifically target tactical networks. This library of applications will be referred to as the *tactical edge suite*. Once developed, the tactical edge suite will be used to actively map and diagnose issues in tactical networks. This chapter will discuss the design philosophy of edge suite applications, as well as the testing methodology and testing platforms used. Individual applications will be presented and their use in tactical data networks will be explained.

4.1   Design Philosophy of Application Suite

Individually, each application is designed to serve a very specific purpose, such as finding nodes or discovering paths. Limiting application scope keeps the overall size of the application small and performance predictable. By combining or even chaining the output of multiple applications together, network operators can incrementally build on information discovered about network state, eventually reaching a live view of the network. Such a view of the network is quite different from the pre-programmed view in applications such as Solar Winds, which are often a representation of what the network *should* look like. By incrementally building a view of the *actual* network state, network operators should be able to make more informed decisions during troubleshooting.

4.2   Assumptions Made for Testing Purposes

In order to allow for testing, assumptions on the breadth of the XPLANE in a network have to be made. In other words, a decision always has to be made on *which* nodes in a network will be XPL-enabled. In a perfect world, all nodes would be part of the XPLANE, as this would allow for the most accurate view of the network from an XPLANE perspective. For the purposes of this thesis, the assumption is that all network infrastructure is XPL-enabled (i.e., routers and switches). In the author's view, this is the most likely deployment strategy for XPLANE in future tactical networks.

With the design philosophy and assumptions in mind, the applications can be introduced. For each application, a number of points will be addressed. First, the rationale behind developing each application, and its relevance in tactical networks will be explained. Next, the application's general algorithm will be explained and presented. Sample use cases will be given, and any notes on performance, behavior, and expected outputs will be given. Finally, the XPL source code will be listed in Appendix A. It is important to note that the source code listings are in uncompiled form; the code must be compiled by the CPS transformation to be used correctly on the XPLANE.

4.3   Discover XPLANE Nodes Using OnFlood

This application utilizes the XPLANE *OnFlood* function to discover all XPL-enabled nodes in the network. This application is designed to be a starting point for gathering network information. By first discovering the XPL-enabled nodes on the network, network operators can build on this information by next discovering links, IP addresses, and other relevant troubleshooting information. The importance of having an accurate view of the network topology in troubleshooting situations cannot be overstated; without an accurate view of the network, network operators are left at a severe disadvantage.

4.3.1   Algorithm Description

The application starts by making an initial call to a recursive function. This function checks the local XPLANE node ID against a list of already visited nodes. If the node ID is not found in the list, it will add the local node ID and flood to all directly connected neighbors. The terminating condition for the application is returning to a node already visited. When this happens, the list of discovered nodes is returned to the originating node. This list not only contains discovered nodes, but also discovered adjacencies between nodes, as the nodes are added in the order they were traversed. The general algorithm is shown as Algorithm 3.

4.3.2   Assumptions

The application assumes there is an XPL-enabled device within the local broadcast domain, which follows from the originally stated assumptions for testing.

Algorithm 3 Discover XPLANE nodes using OnFlood

1: **procedure** DISCOVERNODES(*path*)   ▷ path=list of nodes visited so far
2:      *currentNode ← Node()*
3:      **if** *Member(currentNode, path) = true* **then**
4:          **return** *path*   ▷ path contains all nodes visited
5:      **else**
6:          *path ← AppendToList(path, currentNode)* ▷ Append the current node to path
7:          *OnFlood(DiscoverNodes(path))*
8:      **end if**
9: **end procedure**

4.3.3   Additional Notes

The application returns a list to the originating node, with the order of discovered nodes starting from left to right. The use of *OnFlood* will create multiple copies of the application, which will in turn generate many duplicate results at the originating node. In large networks, the number of copies of the application can grow quite large, and may affect network performance. In other words, this strategy of node discovery is *noisy*. A good use-case for this particular strategy would be a network with unreliable transmission paths. By making multiple copies of the application, the likelihood of receiving path information at the origin increases. For example, for just a single path of length *k* there will *k-1* attempts to report the first hop of the path, *k-2* attempts to report the first two hops, and so on.

4.4   Discover XPLANE Nodes Using Depth First Search

This application utilizes a depth-first search (DFS) approach to discover all XPL-enabled nodes in the network. Just as with discovery via flooding, this application is designed to be a starting point for gathering information on network topology, that, unlike Algorithm 2, generates less traffic. However, it does not produce path information like Algorithm 2 does.

4.4.1   Algorithm Description

This application starts by creating a list of interfaces on the current node. For each interface, it enumerates all directly connected neighbors via XPLANE's *Node.direct* function. For each neighbor, the application relocates and begins the same process again. All node

identifiers discovered along the search are appended to a list, which is eventually returned to the originating node. The general algorithm is shown as Algorithm 4.

Algorithm 4 Discover XPLANE nodes using depth-first search

1: **procedure** INTERFACES(lst, s)
2:      **if** $lst = null$ **then**
3:          **return** s
4:      **else**
5:          $directNeighbors \leftarrow Node.direct(Head(lst))$
6:          $Interfaces(Tail(lst), Neighbors(directNeighbors, s))$
7:      **end if**
8: **end procedure**
9: **procedure** NEIGHBORS(lst, s)
10:      **if** $lst = null$ **then**
11:          **return** s
12:      **else**
13:          $Neighbors(Tail(lst), On(Head(lst), Visit(s)))$
14:      **end if**
15: **end procedure**
16: **procedure** VISIT(s)
17:      **if** $Member(Node(), s) = true$ **then**
18:          **return** s
19:      **else**
20:          $Interfaces(Node.ifaces(), AppendToList(s, Node()))$
21:      **end if**
22: **end procedure**

4.4.2 Assumptions

The applications assumes that XPLANE beaconing is turned on, or that nodes have had sufficient time to learn of neighbors through other means.

4.4.3 Additional Notes

The application returns a list of discovered node identifiers to the originating node. There is no implied adjacency information in the returned list. This application uses the XPLANE *On* function for relocation, and therefore does not create multiple copies of itself in the network. This makes the DFS-based approach less noisy than the flooding approach. It does however increase the chance of failure; if the XPLANE packet is dropped during any

relocation at runtime, the entire computation will be lost. This method of discovery should be avoided in highly unreliable networks. A good use-case for this application would be node discovery in a stable network where a less-noisy discovery process is preferred.

4.5 Enumerate Information on a Remote Node

This application will traverse a known path to a remote node. The path may have been discovered using Algorithm 3. Once at the destination node, the application will enumerate the IP address, subnet mask, MAC address, and directly connected neighbors for each interface. This information is then returned to the originating node as a nested list.

4.5.1 Algorithm Description

The application starts by taking a unidirectional path as input. It then begins relocating to nodes in the path. At each relocation, the application checks to see if the current node ID matches the destination node ID. If it does not, the application relocates to the next node in the path. If it does match, the application calls a recursive function that enumerates information on all node interfaces. Once complete, the information is returned to the originating node. This application has two terminating conditions. The first occurs if the application fails to reach the destination node ID by the specified path. The second occurs when the applications runs out of interfaces to enumerate on the distant node. The general algorithm is shown as Algorithm 5.

4.5.2 Assumptions

As with other protocols, a best effort attempt is given to reach the destination node. Recent changes in network conditions and other factors can affect the outcome of the application. The application will silently terminate if the destination node cannot be reached via the provided path.

4.5.3 Additional Notes

The application returns a nested list to the originating node. A use-case for this application would be to enumerate information on a node ID that is known to exist in the network, such as after running a discovery application.

Algorithm 5 Enumerate information on a remote node

1: **procedure** TRAVERSE(*path*)   ▷ path=list of nodes
2:     **if** $Head(path) = Node()$ **then**
3:         $EnumNodeInfo(List(), Node.ifaces())$ ▷ List() = empty list
4:     **else**
5:         $path \leftarrow Tail(path)$ ▷ Remove first element of path
6:         $On(Head(path), Traverse(path))$
7:     **end if**
8: **end procedure**
9: **procedure** ENUMNODEINFO(*nodeInfoList, devs*)▷ devs=interfaces to be enumerated
10:     **if** $devs = null$ **then**
11:         **return** *nodeInfoList*   ▷ nodeInfoList contains info on all interfaces
12:     **else**
13:         $d \leftarrow Head(devs)$ ▷ d=current interface
14:         $devInfo \leftarrow List(d, Node.ip(d), Node.mask(d), Node.ethaddr(d), Node.direct(d))$
15:         $AppendToList(nodeInfoList, devInfo)$
16:         $EnumNodeInfo(nodeInfoList, Tail(devs))$
17:     **end if**
18: **end procedure**

4.6   Re-positional Ping

This application will traverse a known path to a distant node. Once at the destination node, the application will attempt to ping the IP address given as an argument. The value in this application comes from the ability to test connectivity from the point of view of another XPLANE node. Such information is essential in diagnosing routing issues, firewall issues, and in gaining insight to network flows.

4.6.1   Algorithm Description

The application starts by taking a unidirectional path as input. It then begins relocating to nodes in the path. At each relocation, the application checks to see if the current node ID matches the destination node ID. If it does not, the application relocates to the next node in the path. If it does match, the application calls the *ping* function, which will use XPLANE's packet injection and capture mechanisms to test connectivity with the target IP address. The application will return *true* to the originating node on a successful ping, otherwise it will return *false*. The general algorithm is shown as Algorithm 6.

28

Algorithm 6 Re-positional ping

1: **procedure** TRAVERSE(*path*)   ▷ path=list of nodes
2:   **if** $Head(path) = Node()$ **then**
3:     $Ping(ipAddr, iface)$ ▷ ipAddr=IP to receive ping, iface=interface to send on
4:   **else**
5:     $path \leftarrow Tail(path)$ ▷ Remove first element of path
6:     $On(Head(path), Traverse(path))$
7:   **end if**
8: **end procedure**
9: **procedure** PING(*destIP, dev*)   ▷ dev=network adapter on host to ping from
10:   $Send(dev, List("EchoRequest", destIP));$
11:     $Pcap(dev, "icmp", 3, ProcPkts(pkts, destIP)))$
12: **end procedure**
13: **procedure** PROCPKTS(*pkts, source*)
14:   **if** $pkts = null$ **then**
15:     **return** $false$
16:   **else**
17:     $pkt = Head(pkts)$ ▷ grab next packet in the list of captured packets
18:     **if** $pkt.sourceIP = source$ **then**
19:       **return** $true$
20:     **else**
21:       $ProcPkts(Tail(pkts), source)$ ▷ continue processing list of pkts
22:     **end if**
23:   **end if**
24: **end procedure**

4.6.2   Assumptions

Similar to Algorithm 5, a best effort attempt is given to reach the destination node. Any network changes could affect the outcome of the application. The application will silently terminate if the destination node cannot be reached via the provided path.

4.6.3   Additional Notes

This application is useful for testing connectivity between two distant nodes. The information returned from this application could be extremely important in scenarios such as troubleshooting firewall problems, access control list problems, or general connectivity issues between two known IP addresses that *should* be able to communicate. The real value in these scenarios is getting the network to perform actions that usually require physically

re-locating an administrator. By automating the test from a distance, a network operator need not rely on another person at the distant node.

4.7 Testing Platform for the Tactical Edge Suite

The approach used to test applications in the tactical edge suite is commonly referred to as *unit testing*. In unit testing, code is executed on a specific input, in a specific environment, and the output of that code is compared against expected results. In the case of the tactical edge suite, the code was each individual application, and the environment was a series of different networks, all configured in a way to specifically test that application. Special attention was paid to edge cases in order to verify the behavior of applications.

4.7.1 CORE

To facilitate testing, the Common Open Research Emulator (CORE) virtualization platform was used. CORE is a tool developed by Boeing Research and Technology and supported by the Naval Research Laboratory (NRL) for emulating networks [16]. Networks built in CORE are completely virtual; the network infrastructure and nodes all run as virtual machines that emulate the nodes they represent. Figure 4.1 shows an example network designed in CORE.

Each router, switch, and host can be configured to run different built-in protocols and services, as seen in Figure 4.2. The interface is largely a drag-and-drop, Visio-style interface that has a small learning curve for experienced network designers.

The CORE platform offers three important features for testing. The first feature is the ability to communicate with the virtual networks. Networks built in CORE can be *bridged* with real networks, allowing communication between physical and virtual nodes. This allows application development to take place on a physical machine while still allowing testing with a virtual network. The second feature is CORE's ability to induce errors into the network. Figure 4.3 shows CORE's link configuration menu. CORE allows users to artificially introduce problems such as latency, dropped packets, and duplicate packets into network links. This feature is crucial for emulating the performance of XPLANE applications in malfunctioning networks. The third feature is CORE's built-in *Topology Generator*, which allows users to rapidly generate a network of nodes from a menu-driven
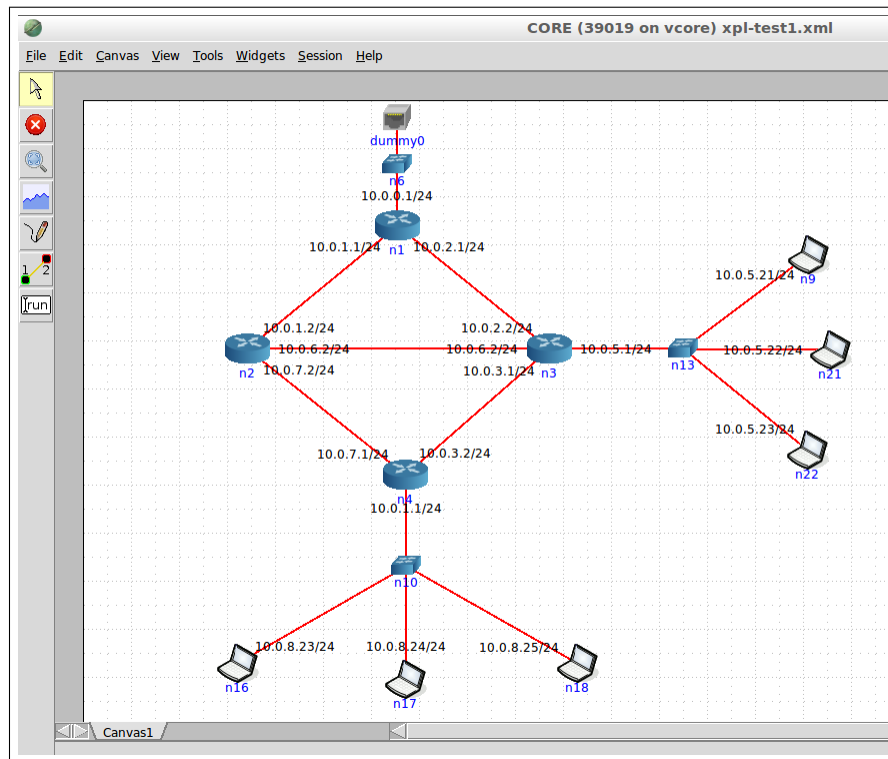
Figure 4.1: An example virtual network running in CORE

interface. Users can select from a number of common topologies to include star, grid, cycle, and clique. CORE even has the option to generate random topologies. The CORE Topology Generator was key to rapid testing in multiple topologies. CORE has built-in Python scripting support, which allows for scripting of common tasks. CORE also comes with a internal debugger, and various interfaces to modify the internals of CORE itself.
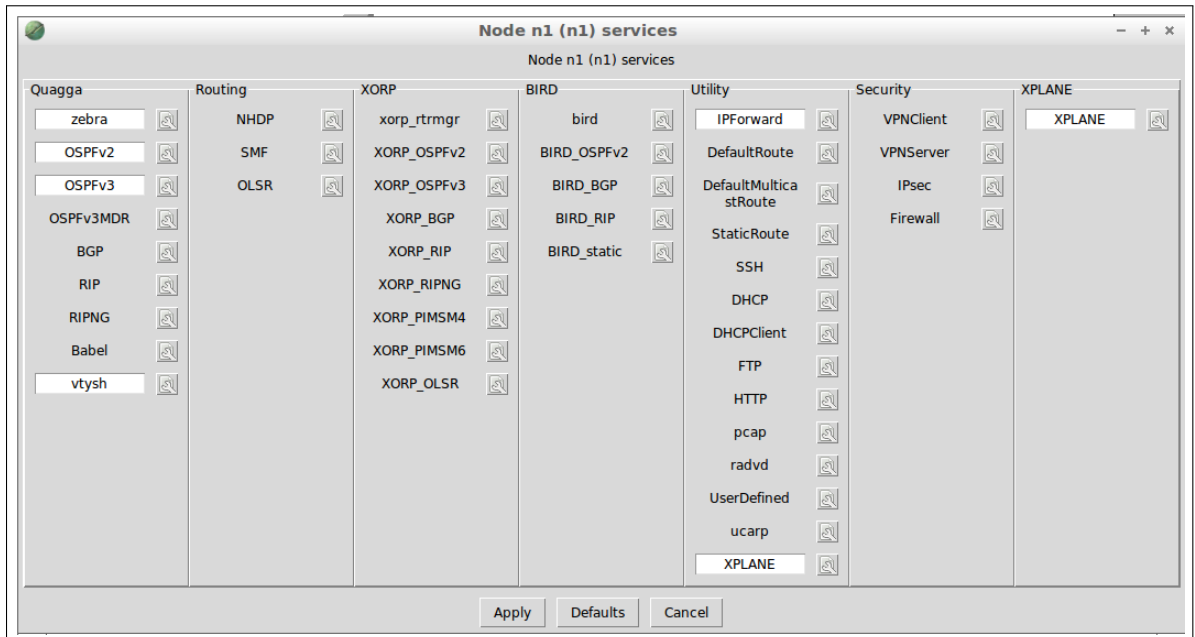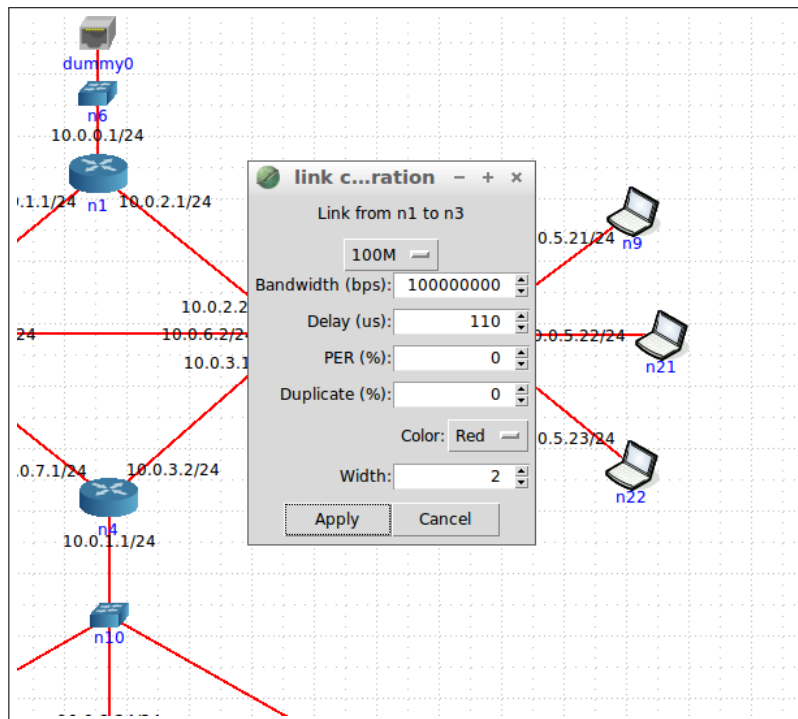
Figure 4.2: CORE node configuration window

Figure 4.3: CORE link configuration menu

CHAPTER 5:
Developing an XPLANE server and Web-based User Interface

In order to keep the state of results from tactical edge suite applications, an XPLANE server has been developed. The XPLANE server maintains state about a network and provides a basic command line interface for injecting code into the XPLANE and capturing results. A web-based UI has also been developed and interfaced with the XPLANE server, which gives network operators a detailed view of the network topology.

5.1   XPLANE Server

In its current state, XPLANE consists of XPL and the runtime environment [7]. XPLANE applications are launched via a command line program that accepts XPL-scheme and injects the code into the local XPLANE node. Results of computations can be "seen" by viewing the output from the XPLANE shim on the node where the application terminates. On termination, the results are not stored in the XPLANE; it is up to the user to capture, remember, and make sense of any information returned. This design is efficient for the network and XPLANE runtime environment, but makes life harder for the end user.
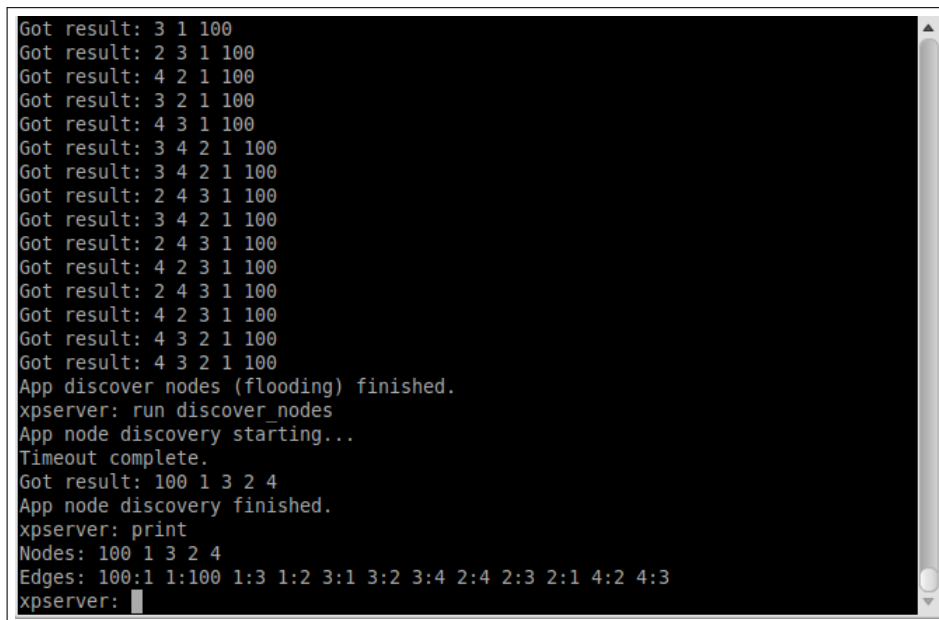
Remembering and interpreting results are not the only usability issues for network operators. Applications from the tactical edge suite often expect certain parameters or *arguments* to be defined, such as target node identifiers or IP addresses. Each time an application runs, the code needs to be hand-modified with the correct arguments. If the code is expected to return results from a distant part of the network, it also needs to be recompiled with the CPS transformation [7]. This requires users to know how to edit XPL code and compile programs with the CPS transformation. These barriers to usability have been overcome by designing an *XPLANE server*.

The goal of the XPLANE server is to provide a layer of abstraction between the XPLANE runtime system and the network operator. The server is responsible for launching applications from the tactical edge suite, capturing any results, and saving the overall network

state. The main XPLANE server components are the command-line interface, a set of classes representing the tactical edge suite, and an embedded web server. The XPLANE server is written in the Java programming language and is therefore platform-agnostic. The XPLANE server requires nothing more than the current Java Runtime Environment, which is available on most modern platforms. All necessary support libraries are bundled with the server itself.

5.1.1 Command-line Interface

The command-line interface allows users to run edge suite applications and view results, display and modify network state, and push manual updates on network state via the web server. Communication of network state to clients will be addressed in Section 5.1.3. Figure 5.1 shows a screen capture of the server's command-line interface after both running an application and printing out a summary of the network state. . The command-line interface can be thought of as an XPLANE shell; it will infinitely loop, accepting commands and displaying their output until the server is explicitly stopped by the user.

```
Got result: 3 1 100
Got result: 2 3 1 100
Got result: 4 2 1 100
Got result: 3 2 1 100
Got result: 4 3 1 100
Got result: 3 4 2 1 100
Got result: 3 4 2 1 100
Got result: 2 4 3 1 100
Got result: 3 4 2 1 100
Got result: 2 4 3 1 100
Got result: 4 2 3 1 100
Got result: 2 4 3 1 100
Got result: 4 2 3 1 100
Got result: 4 3 2 1 100
Got result: 4 3 2 1 100
App discover nodes (flooding) finished.
xpserver: run discover_nodes
App node discovery starting...
Timeout complete.
Got result: 100 1 3 2 4
App node discovery finished.
xpserver: print
Nodes: 100 1 3 2 4
Edges: 100:1 1:100 1:3 1:2 3:1 3:2 3:4 2:4 2:3 2:1 4:2 4:3
xpserver:
```

Figure 5.1: Screen capture of the XPLANE server's command-line interface

5.1.2 Edge Suite Classes

In order to integrate the tactical edge suite in to the server, each individual application is "wrapped" inside a Java class. Each class consists of a string representation of the CPS compiled XPL code, a constructor method, and a *run()* method. The XPL code has been modified by removing all application-specific parameters and replacing them with Java string format specifiers. The constructor is responsible for taking arguments passed via the command line and formatting the XPL code string with the given arguments. The *run()* method is where the work is done. When called, *run()* will inject the modified XPL code into the local XPLANE shim. It will then begin capturing output from the XPLANE shim and look for results from the injected application. However, there may be no results if the network experiences a fault that prevents the application from completing execution.

Dealing with Incomplete Executions

When designing a Java wrapper class to an XPLANE application, an important design decision is how to cope with partial executions due to network issues. One reason for partial executions is the mechanism of relocation in XPLANE. Applications relocate via Ethernet frames, and if a frame carrying an XPLANE application is dropped during transmission, the computation will be lost. By design, the XPLANE does not "keep state" of computations in the network; it is up to each individual computation to keep any necessary state. For these reasons, a Java wrapper class needs to make a determination on how long it is willing to wait to receive results from an XPLANE application. After this *timeout* the wrapper class can either report application timeout or possibly re-run the XPLANE application. All of these decisions can either be left to the wrapper class or the user can be queried for what to do when these situations occur.

One of the most important functions of the *run()* method is receiving XPLANE application results and parsing them accordingly. This operation often needs to be tailored for the specific XPLANE application it is wrapping, which is why a generic container for XPLANE applications cannot be used. Once the results have been received and parsed, the *run()* method is responsible for updating the server's internal network state. Once the *run()* method returns, the XPLANE server may optionally push network status updates to any connected web clients.

5.1.3   Embedded Web Server

The XPLANE server comes with an embedded Hypertext Transfer Protocol (HTTP) server known as Jetty. Jetty is an open source web server and Java Servlet container, that also supports a number of other web technologies including WebSockets [17]. Jetty is written in Java and maintained by the Eclipse Foundation. Jetty can run as a standalone web server or be embedded within a larger application. In the case of the XPLANE server, Jetty is run in embedded mode. When the XPLANE server is started, it launches an embedded Jetty instance within itself to serve web content to clients. Jetty was chosen for use with the XPLANE server due to it's mature implementation of the WebSocket protocol.

WebSockets

The WebSocket protocol is a relatively new Internet Engineering Task Force (IETF) standard. The goal of the protocol is to provide browser-based applications a way to create a persistent, full-duplex HTTP connection with a server [18]. The traditional client-server model does not work well with applications that require sporadic updates from the server to the client. These issues have been traditionally handled with workarounds such as 'long polling' in which the client continually polls the server for information updates [18]. With WebSockets, clients can initiate the connection and simply wait for the server to *push* updates vice constant polling from the client side.

Once the WebSocket connection is complete, the state of the network as determined using the XPLANE is pushed to the newly connected client. This state information consists of network nodes, edges between nodes, and metadata on nodes such as IP addresses, Ethernet media access control (MAC) addresses, and node type. This data is formatted in JavaScript Object Notation (JSON). Even though JSON is primarily used in JavaScript applications, it is a language-independent data format and can be interpreted by many programming languages. After the state of the network is initially pushed, the WebSocket connection will persist until the client ends the connection. Any further network state updates will automatically be pushed to all connected clients.

5.2   Web-based User Interface

In order to visualize the network state provided by the XPLANE server, a web browser-based UI was created. Using a web browser as the interface was chosen due to the rich en-

vironment of technologies that modern web browsers offer including JavaScript, WebSockets, and Hypertext Markup Language Version 5 (HTML5). A web-based UI that focuses on open standards such as these helps remove the need for platform-specific requirements. This will allow the UI to run on most modern systems without modification.

5.2.1 Design Overview

At its core, the UI is a JavaScript application. When the UI connects and receives the initial network state, it parses the JSON data into internal array objects. It then feeds these arrays to a library known as Data-Driven Documents (D3). D3 is a JavaScript library for visualizing data sets using HTML5, Scalable Vector Graphics (SVG), and Cascading Style Sheets (CSS) [19]. More specifically, D3 allows data to be transformed into graphical objects in the web broswer that can be interacted with and later modified via the Document Object Model (DOM). For the purposes of this UI, the network state is rendered with D3 as a *force-directed graph*. Force-directed graphs turn networks of nodes into a physical simulation by assigning certain properties such as charge, friction, and gravity to nodes. It then allows these nodes to interact in such a way that the graph naturally finds an equilibrium. At this equilibrium, the nodes will have largely separated and un-clustered from each other, presenting a more readable graph. Figure 5.2 shows an example force-directed graph generated from XPLANE server data. By using a force-directed layout, we escape the issue of figuring out how to render the graph manually, which is a non-trivial problem to solve.

5.2.2 Nodes and Graph Interaction

Nodes in the graph are represented by different graphics, depending on the node type specified in the JSON data. Routers display as the familiar router network icon, hosts display as computer monitors, and nodes of type *unknown* display as questions marks. The graph allows nodes to be rearranged using the mouse; all other nodes in the graph will respond to changes in the force layout and react accordingly. This can be useful if the simulation failed to find a suitable equilibrium. Nodes will respond to the *mouse-over* event by displaying a pop-up menu. The pop-up menu will display all information available on a node. Figure 5.3 shows an example pop-up for a test network.
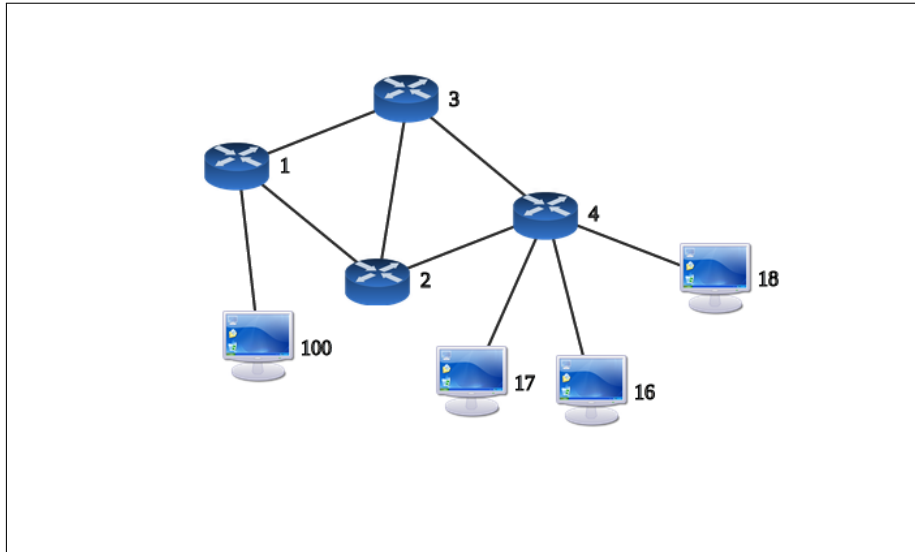
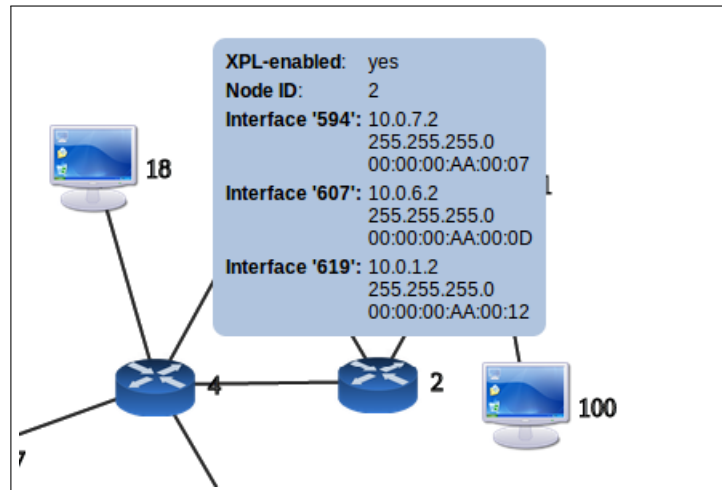Figure 5.2: An example force-directed graph generated from XPLANE server data

Figure 5.3: UI pop-up window on node mouse-over event

5.2.3 Server Updates

Once the UI has successfully downloaded, parsed, and rendered the JSON network data, the WebSocket connection created with the XPLANE server will remain open. Any changes in network state will automatically be pushed from the server to all connected clients. When the UI receives an update, it will parse the new JSON data and compare the new data to the old network data. New nodes and connections will be added to the graph, and deleted nodes and connections will be removed. This method of updating the graph prevents D3

from completely re-rendering the force-directed layout from scratch on every update, which would make the graph unstable. Nodes and connections that persisted between updates will stay relatively close to their original positions and will adjust only for new or deleted nodes.

5.2.4 Anomaly Detection

In addition to rendering the network visually, the UI will also attempt to detect anomalies in the network. Anomaly detection occurs outside of the XPLANE by processing the raw JSON network data in the UI and looking for known network anomalies. At this time, the only anomaly successfully detected is duplicate IP addresses. Detected anomalies are displayed in a table within the UI. Figure 5.4 shows a screen capture with two detected network anomalies. Anomaly detection is run every time a network update is received, and the detected anomalies table is updated accordingly.

Anomaly	Time	Nodes Involved
Duplicate IPs	04/16/2014 13:05 PM	1,4
Duplicate IPs	04/16/2014 13:05 PM	3,2

Figure 5.4: Screen capture of detected anomalies table

THIS PAGE INTENTIONALLY LEFT BLANK

CHAPTER 6:
XPLANE Case Study

In order to compare the utility of the XPLANE toolset presented in Chapters 4 and 5 against traditional tools, a case study was conducted. For this study, a battalion-sized tactical data network was constructed in a laboratory setting. The case study is focused around the scenario of troubleshooting a geographically dispersed network from a central location much like the author's experience in Afghanistan. First the network used for the case study will be presented. Next an attempt to troubleshoot the network using traditional tools and protocols will be presented. This will be followed by an attempt to troubleshoot the same scenario using the XPLANE toolset presented in this thesis. Finally the results of both troubleshooting attempts will be compared.

6.1 Network Layout

The network consists of four notional company positions: Alpha, Bravo, Charlie, and Headquarters. Each company position consists of a router, a switch, and two workstations. The number of workstations was kept small for simplicity. The company positions are connected through a number of means. Headquarters is directly connected to all three companies. Alpha and Charlie company are notionally located in the same geographic area and share a redundant link between each other. Bravo company is geographically isolated from the other companies. All companies are notionally operating at a significant distance from Headquarters company; travelling to other company positions for troubleshooting is a last resort. Each company is responsible for a class C subnet. Static routing was utilized due to the static nature of company positions. Figure 6.1 shows a diagram of the network architecture.

Once the network was functioning correctly, errors were introduced at the logical layer to emulate a malfunctioning tactical network. The errors introduced were common misconfigurations inspired from the author's operational experience. The network has been logically broken in three ways:

1. The routing table on Alpha company's router has been accidentally deleted during a
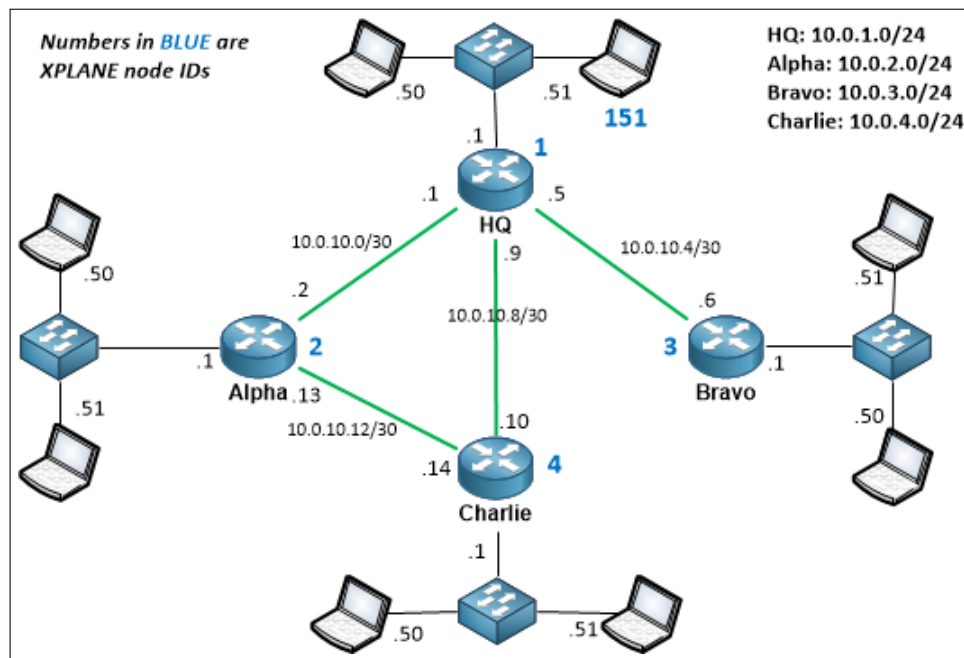
Figure 6.1: Network diagram for case study before errors are introduced

hypothetical troubleshooting session.

2. Bravo company's router has an incorrect subnet mask specified for the link to Headquarters company.

3. Charlie company's router has an incorrect (and duplicate) IP address on the interface connected to Headquarters company.

A complete description of the network configuration and induced errors can be found in Table B.1 in Appendix B. No virtualization was employed for the case study; all nodes are physical. The tactical transmission systems that would connect company positions in a real world scenario are notional. Ethernet cables were used to connect company positions in the laboratory setting. For the purposes of troubleshooting we will assume that the notional transmission systems have been verified as working correctly. Each router is a Linux-based Soekris router. The XPLANE shim requires a Linux-based environment at this time, which Soekris routers provide. The XPLANE shim is running on all four routers as well as the troubleshooting workstation in the Headquarters subnet. The node identifiers for each XPLANE node are displayed in Figure 6.1. The XPLANE shims have beaconing enabled. All troubleshooting will take place from the XPL-enabled node within the Headquarters

network (XPLANE node ID 151).

6.2   Troubleshooting with Traditional Tools

The first troubleshooting attempt was made with the tools and protocols described in Section 2.3. For each tool, a description of the way it was employed as well as the results of all tests performed will be explained.

6.2.1   Ping

The first step in network troubleshooting is to test logical layer connectivity between hosts. In this case, the connectivity between Headquarters and all three companies is in question. The first tool often employed for this test is the ping program or some version of it. For this case study, we will utilize the program *Nmap* to automate a network-wide ping scan on our behalf. Nmap will send a ping packet to every host in the specified networks and report back any replies. Even though we have a good idea of which hosts to ping from the network documentation, we will tell Nmap to scan the four class C subnets in use with the following command: *nmap -sP 10.0.1.0/24 10.0.2.0/24 10.0.3.0/24 10.0.4.0/24 10.0.10.0/24*. Nmap produced the following output:

```
Nmap scan report for 10.0.1.1
Host is up (0.00081s latency).
Nmap scan report for 10.0.1.51
Host is up (0.00028s latency).
Nmap scan report for 10.0.10.1
Host is up (0.00044s latency).
Nmap scan report for 10.0.10.5
Host is up (0.00098s latency).
Nmap scan report for 10.0.10.9
Host is up (0.0015s latency).
Nmap done: 1280 IP addresses (5 hosts up) scanned in 20.88 seconds
```

Looking closely at the live IP addresses we quickly notice that the only replies received were from our own workstation and the interfaces on the Headquarters router. This adds evidence to the possibility that logical layer connectivity is down with all three companies

for some reason. It really only confirms that we cannot send ICMP traffic to the company networks; it says nothing for other logical layer protocols. The next step would be to verify all settings on the Headquarters router, which would produce no answers since it correctly configured. In order to confirm that the issue affects the entire logical layer, we can use Nmap again to perform a TCP-based ping. This will make Nmap attempt a TCP connection to a specific port to see if any response is received at all. The following command was used: *nmap -PS22 10.0.2-4.1*. This tells Nmap to send a TCP connection request to port 22 of each company router and look for any reply. In our case, port 22 of the routers is running secure shell (SSH) and therefore *should* reply. Nmap produced the following output:

```
Nmap done: 3 IP addresses (0 hosts up) scanned in 2.07 seconds
```

It is now clear that the logical layer is broken in some way. Unfortunately we have exhausted the utility of connectivity testing tools such as ping and must move on.

6.2.2 Traceroute

Traceroute is used in troubleshooting to discover logical paths between two nodes. Traceroute will show each hop along the path, and more importantly, the *last successful* hop. If the last hop is not the target node, then we conclude that the source of the problem lies somewhere close to the last successful hop along the path. In our case, traceroute will be of little value. We have already shown that the logical layer is malfunctioning, which traceroute relies on. On top of that, we are able to validate our own router's configuration which appears to be error-free. Regardless, it is never a bad idea to try all available tools. Upon running traceroute on *10.0.2.1*, *10.0.3.1*, and *10.0.4.1*, the tool confirmed the last hop before failure is the Headquarters router. This at least confirms that traffic is making it from our workstation to the Headquarters router. Beyond that, no other information is provided for our troubleshooting efforts.

6.2.3 SNMP

SNMP can often be used in troubleshooting situations as long as network devices support the protocol. In our case it makes no difference. SNMP is a UDP-based protocol that lives at the logical layer of the network. Until we can fix the logical layer, tools that exist at this layer will be of limited value.

6.2.4   Link Layer Discovery Protocols

Protocols such as Link Layer Discovery Protocol (LLDP) cannot be applied to help diagnose problems in routed networks. Our case study network is segmented into different class C subnets for each location. Link layer discovery protocols are limited to the local subnet by definition. They cannot help in troubleshooting problems across subnet boundaries.

6.2.5   Results with Traditional Tools

We were able to gather evidence that there are logical layer connectivity issues to all three companies. We are able to verify that the Headquarters' router configuration is correct. Combined with the previous assumption that transmission systems are working correctly, we conclude that there are logical configuration issues on all three company routers. We have no idea *what* the issues are and therefore need an experienced network operator to troubleshoot each router configuration. If we do not have a seasoned network operator at each location we would have to physically send one.

6.3   Troubleshooting with XPLANE

The second troubleshooting attempt was made with the tools presented in Chapters 4 and 5. As a reminder to the reader we are XPLANE node ID 151 and the XPLANE shim is running on all routers. We begin by starting the XPLANE server and connecting the web-based UI. At this point the only thing we *know* about in the XPLANE is ourselves. Figure 6.2 shows our current view of the network.

Figure 6.2: Initial view of the XPLANE during troubleshooting

6.3.1   Node Discovery with Depth-first Search

We start by attempting to discover nodes in the network using the depth-first search discovery application discussed in Section 4.4. Figure 6.3 shows our view of the network after depth-first search discovery.
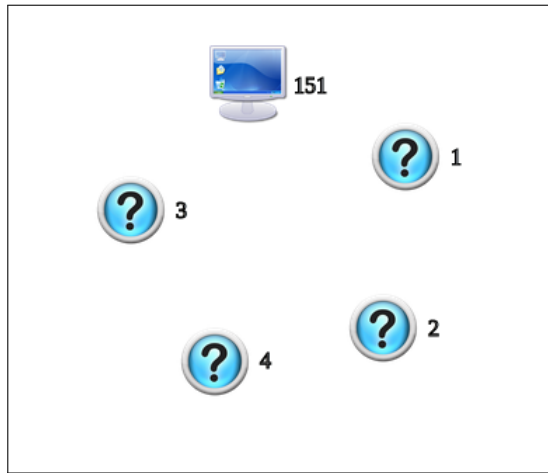
Figure 6.3: View of the XPLANE after node discovery

We discovered four additional nodes numbered one through four which aligns with our network documentation. At this point we would like to query each node for information but we do not know *how* each node is connected. We have an additional discovery application in our toolbox that we can use.

6.3.2 Path Discovery with OnFlood Search

The discovery application described in Section 4.3 gives us both node and path information, so we will use that next. Figure 6.4 shows the network view after path discovery.
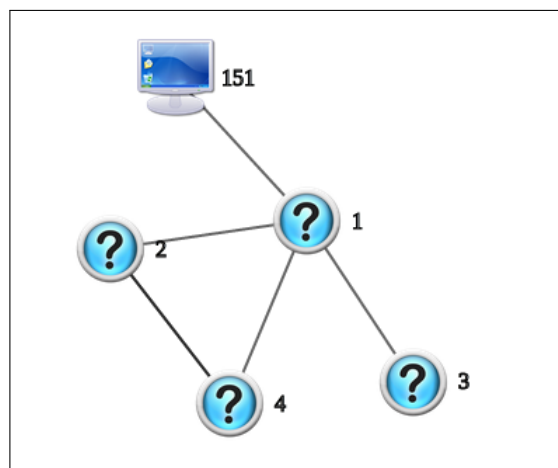

Figure 6.4: View of the XPLANE after path discovery

6.3.3 Router and Host Enumeration using XPLANE

The network topology now looks similar to what was expected. We continue by using the application described in Section 4.5 to enumerate information from each node. Figure 6.5 shows the network after enumerating nodes one and two.
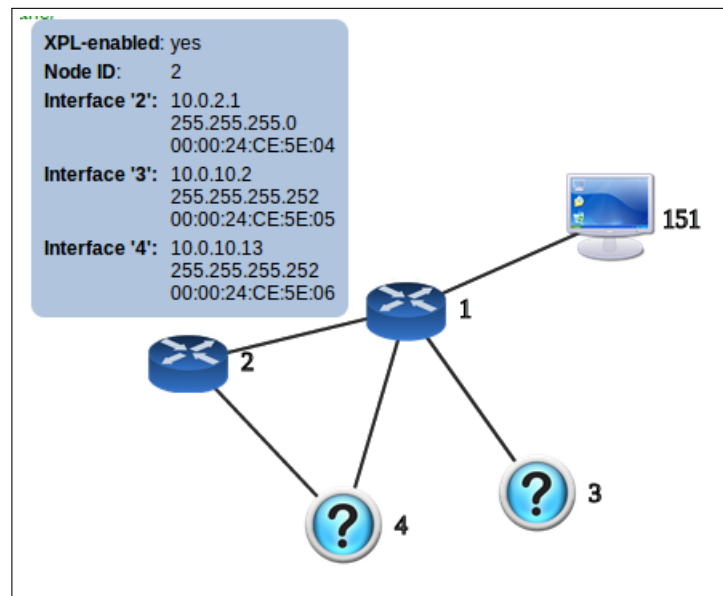

Figure 6.5: View of the XPLANE after enumerating first two nodes

The information pop-up shows all enumerated information for node two. Node two is clearly Alpha company's router. Now that we have some basic information on its live configuration we can begin to troubleshoot. Comparing the live configuration to the network documentation yields no discrepancies. Yet for some reason we still cannot communicate with Alpha company which suggests the issue lies deeper. Unfortunately XPLANE does not currently provide any additional information on Alpha's router. We can however run XPL applications on Alpha's router, which is what we will do next. We will attempt to discover the two hosts on Alpha's subnet using the re-positional ping program described in Section 4.6. The program will attempt to ping the IP addresses of the hosts from Alpha's router and report back the results. Figure 6.6 shows the results of our ping attempts.
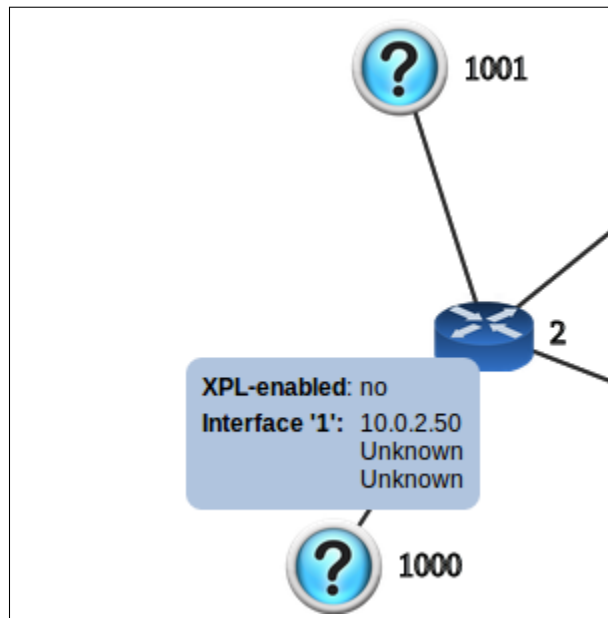
Figure 6.6: View of Alpha's network after discovering hosts

We have discovered two additional nodes that have been assigned node identifiers 1000 and 1001. Since XPLANE is not running on edge devices, they have no XPLANE node identifier. The XPLANE server makes up for this by generating a unique node identifier for nodes that lie outside the XPLANE. The current numbering scheme starts at 1000 and increases by one for each new node found. The number can be anything *as long as it is unique*. The information pop-up shows all enumerated information for node 1000 which really only consists of the host's IP address. The identified information shows that these are in fact the two hosts we expected to find in Alpha's network. We will continue with the same process for both Bravo and Charlie's subnets. Figure 6.7 shows the graph view of node four as well as the anomaly table after enumerating node four.

Once enumerated, it becomes clear that Charlie's router has a wrong (and duplicate) IP address on its connection with Headquarters. The anomaly was detected by the web-based UI and pointed out in the anomaly table. Charlie's connection to Headquarters should have an IP address of *10.0.10.10*; the last octet is incorrect. We next enumerate the hosts in Charlie's subnet in the same manner as Alpha's. Figure 6.8 shows the results of enumeration.

48
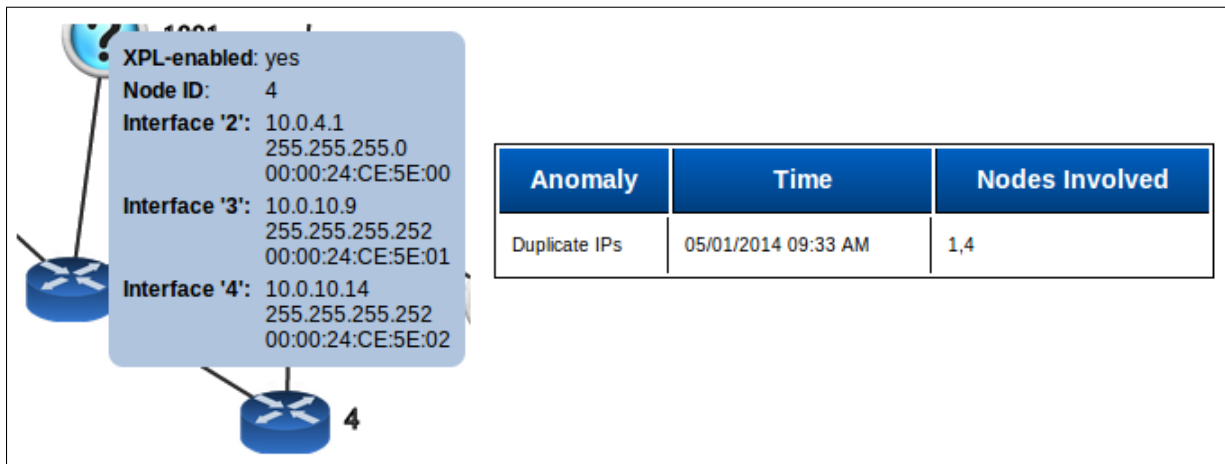
Figure 6.7: View of XPLANE after enumerating Charlie's router
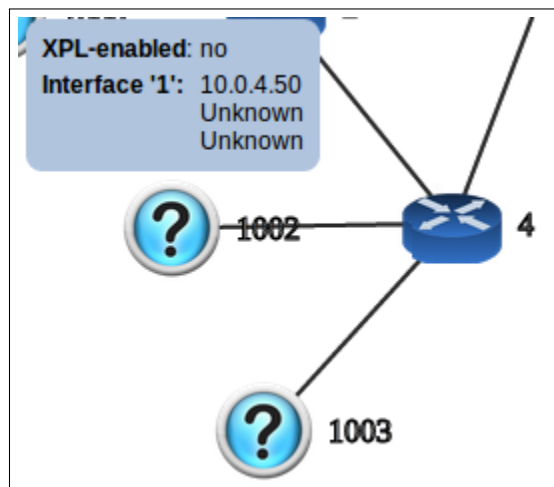
Figure 6.8: View of Charlie's network after dicovering hosts

We have successfully discovered the two hosts in Charlie's subnet, which are now labeled as nodes 1002 and 1003. As with the hosts in Alpha's network, we will be unable to enumerate anything besides the hosts' IP address. We then turn to enumeration of Bravo's subnet. Figure 6.9 shows the result of enumerating Bravo's router.

There are no new entries in the anomaly table. At this time the web-based UI can only detect duplicate IP addresses, so we will need to look deeper to find problems with Bravo's router configuration.

Figure 6.9: View of XPLANE after enumerating Bravo's router

Upon inspection of the information provided by XPLANE, we notice an incorrect subnet mask on Bravo's connection to Headquarters. The subnet mask is *255.255.255.254* which is incorrect. The last octet should be 252. This could explain the lack of logical connectivity to Bravo company. We complete the enumeration by discovering the hosts in Bravo's subnet. Figure 6.10 shows the result of enumerating Bravo's subnet and gives our final view of the network as discovered by XPLANE-based tools.
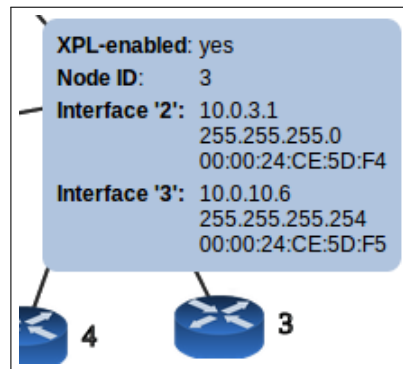
6.4   Comparison of Troubleshooting Attempts

Troubleshooting with traditional tools in this case yielded little information. Troubleshooting with XPLANE-based tools yielded the entire network topology. Our XPLANE-based tools also yielded the configuration of all network interfaces (with the exception of routing) on XPL-enabled nodes. With the first attempt, we were able to rule out the Headquarters router configuration as a source of the problem but were unable to go beyond that. Without remote access to company routers we were unable to continue without relying on the physical presence of a trained network operator at each location. With the information provided by the second troubleshooting attempt we were able to discover an incorrect and duplicate IP address on Charlie's router as well as an incorrect subnet mask on Bravo's router. Fixing these two issues remotely would fix the logical layer connection between Headquarters, Charlie, and Bravo. We were unable to diagnose the connection problems with Alpha. We could attempt to use the redundant link between Charlie and Alpha company once Charlie's connection to Headquarters is fixed, but we would still find that we can not connect. We have reached the limit of what this application suite can achieve. How it might be extended

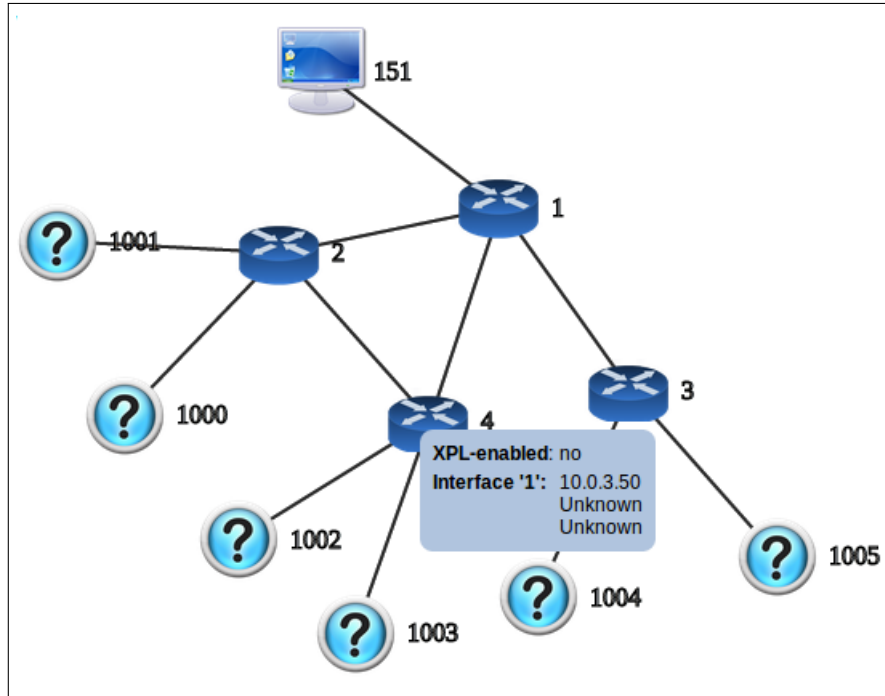with another application to diagnose Alpha's connection problem is addressed in Section 7.2.1.

Figure 6.10: View of the entire network as discovered by XPLANE

THIS PAGE INTENTIONALLY LEFT BLANK

CHAPTER 7:
Future Work and Conclusions

The work presented in this thesis only begins to scratch the surface of what is possible with an XPLANE-like system. Both XPLANE and the XPLANE-based tools developed in this thesis could benefit from additional capabilities and work. We will first address XPLANE and the future capabilities we would like to see. Next we will address the XPLANE-based tools presented in Chapters 4 and 5 in the same manner. Finally we will present conclusions to this research.

7.1 Future Work on XPLANE

XPLANE is an ongoing project. Development efforts have focused on the design of XPL and ensuring the core capabilities of the system function correctly [7]. We will now address capabilities that would increase the utility of XPLANE in tactical networks.

7.1.1 Security

The designers of XPLANE always considered security to be future work [7]. A security architecture is driven by a threat model and without such a model, arguing a system is secure amounts to secure by definition. Different threat models impose different sets of demands on the system. We will define a threat model based on the types of networks discussed in this thesis (i.e., tactical networks) and explore security concerns and mitigations.

Threat Model

The networks considered in this thesis are tactical networks in a field environment. The most likely points of entry to the XPLANE in such networks would be through a device on the local network, open switch ports, or the web-based UI to the XPLANE server. These are the points of access we will consider for our threat model.

Consider a network that is XPLANE enabled but has no mechanism for access control to XPLANE at the Ethernet frame level. Any host on the network could inject arbitrary XPL code into the XPLANE. If the code is poorly written it could have unintended errors such as failure to terminate. Consider the following program:

Algorithm 7 A non-terminating program using OnFlood

1: **procedure** FOO
2:     $OnFlood(Foo())$
3: **end procedure**
4: $Foo()$

Algorithm 7 is an example of a program that fails to terminate. The program additionally uses *OnFlood* for relocation and as a side effect generates many copies of itself. The combination of no terminating condition with flooding will cause this program to eventually consume all available network bandwidth. This example illustrates the need for access control to XPLANE at the frame layer of the network to protect against both naive users and malicious insiders. Naive users can be considered anybody on the network who has the capability to either accidentally inject code or purposefully inject untested code. Malicious insiders can be considered anybody who wishes to use XPLANE in unintended ways such as to disrupt network performance. Hence access control at the frame level is needed.

Access Control at the Frame Level

By implementing access control at the frame level we can restrict code injection to users of the XPLANE server. XPLANE has experimental support for access control at the frame level using message authentication codes (MACs). MACs are a cryptographic technique used to *authenticate* messages between a sender and receiver [20]. MACs are typically functions with two arguments that take the form:

$$MAC(K, m)$$

where *K* is a fixed size cryptographic key and *m* is the message to be authenticated. The output of the MAC function is a fixed size value that is unique to the combination of *K* and *m*. When the sender wishes to authenticate message *m* to the receiver, the sender will send *m* as well as *MAC(K,m)*. It is assumed that the sender and receiver have already agreed on the secret key *K* through other means. Upon receipt of the message, the receiver calculates *MAC(K,m)* and compares it to the MAC value sent with the message. If it matches, the receiver concludes the message has not been tampered with. If it does not match, the receiver discards the message. As long as an attacker does not know the secret key *K*, they cannot change the contents of authenticated messages.

The XPLANE currently has support for HMAC-SHA-1 [21]. HMAC is a keyed hash used for message authentication. It can be used with any one of several cryptographic hash functions. In the XPLANE, it is used with the Secure Hash Algorithm (SHA-1) hash function. The HMAC-SHA-1 function takes as input a secret key K and a message m where m consists of the XPLANE header and payload. When HMAC is enabled, XPLANE will attempt to authenticate XPLANE packets before running the marshaled code contained within the packet. If the packet does not authenticate, the XPLANE shim discards the packet.

This is a good start for access control at the frame level. The next piece would be protection from *replay attacks*. In a replay attack, the attacker takes advantage of the fact that both the message and MAC can be eavesdropped, recorded, and later resent to the original receiver of the message. The notion of *unique message numbers* is often used to prevent replay attacks in protocols that use MACs for authentication [20]. Implementing a message numbering system in XPLANE could prevent replay attacks.

It is worth noting again that MACs only provide authentication. MACs by themselves do no provide any form of *confidentiality*. A malicious insider could eavesdrop on the results of XPLANE applications and learn information about the network they were not intended to have. In a different threat model this could be an issue.

Use of HMAC addresses the threat of unauthorized users injecting code into the XPLANE, providing these users can be prevented from getting the HMAC key. If this can be guaranteed somehow then access to the XPLANE reduces to accessing it through the XPLANE server. Therefore, steps must be taken to limit access to the XPLANE server to authorized users only.

Access Control at the XPLANE Server

The mechanisms for access control at the XPLANE server level are aimed at limiting user access. The XPLANE server should be augmented with user authentication mechanisms. Authentication could be integrated with pre-established user credentials such as those stored on an existing Active Directory server. Users would authenticate via web forms before gaining access to the live view of the network. Once authenticated, users would be restricted to a set of verified applications such as the tactical edge suite. These applications would be guaranteed to perform in a predictable manner and always terminate.

Restricting users to a set of verified applications solves the problem of dealing with non-terminating code; however, it could introduce a new problem. By centralizing control of both XPLANE and the XPLANE server we have given up a key capability of XPLANE. We are now relying on a logical layer protocol (i.e., HTTP) to communicate with the XPLANE server and inject code into the XPLANE. Since XPLANE is meant to be used in situations where the logical layer is malfunctioning, we run the risk of cutting off XPLANE access to users that have no logical path to the XPLANE server. This is one reason that centralizing the XPLANE server might not be the best solution. There are likely others. If the set of applications available to users cannot be controlled then administrators need another option for dealing with non-terminating code.

Time to Live Values

One possible mechanism is to limit the life of an XPLANE program in the network via a TTL value. The idea of a TTL is not a new one. The IP protocol implements TTL values as an integer that decrements at each hop. When the TTL reaches zero, the packet is discarded. This prevents IP packets from traversing the network indefinitely [1]. The same mechanism could be applied to XPLANE packets. Upon injection into the network, the TTL of a program's XPLANE packet would be initialized to a default value. With each relocation in the network the TTL would be decremented until it reaches zero. Any new copies of the program generated with constructs such as *OnFlood* would inherit the current TTL value of the program. This ensures that programs cannot circumvent the TTL simply by producing new copies of themselves. While this neatly takes care of the non-termination problem, it introduces new complexities for XPL application writers. Consider algorithm 3, which discovers network topology using *OnFlood*. In large networks, this algorithm could possibly exhaust the TTL value of the program, forcing early termination. In the program's current form, the user would have no idea whether the program terminated naturally or prematurely. This might leave parts of the network undiscovered if there is no way to alert the user to early termination. One possible solution would be to expose the TTL value of XPL programs to the program itself. This would allow XPL application writers to query the TTL value before or after relocation. This information could be used to make a determination on what to do when the program's TTL begins to approach zero. Possible actions would be to alert the user to the forced early termination, and possibly return partial results of the computation. If the program returns results, it would additionally have to keep

track of the number of hops that would be required to return back to the originating node, as these hops will count against the TTL.

Key Management

HMAC cryptographic key management has many issues. They include generation, rollover induced by network speed, expiration due to policy or compromise, and distribution. Standards such as the Federal Information Processing Standard (FIPS) 140-2 come into play, and the generation, destruction, and accountability of keying material would likely fall under the Electronic Key Management System (EKMS). These issues are beyond the scope of this thesis.

7.1.2   Injection of TCP and UDP Packets

XPLANE currently only supports injection of ICMP packets. Adding support for both TCP and UDP would allow for a broader range of measurements. XPLANE applications could be written to troubleshoot specific ports and application layer protocols by mimicking their behavior at the logical layer and observing network behavior.

7.1.3   Querying a Node's Routing Tables

XPLANE has no built-in support to view a node's routing tables. The case study presented in Chapter 6 is a good example of a troubleshooting situation that could have benefited from such support. The ability to query a distant node's routing tables would open the door to detecting anomalies in routing.

7.1.4   Querying a Node's Management Information Base

If XPLANE is able to find a physical path to an XPLANE-enabled node it could attempt to query the node's MIB for relevant troubleshooting information. This information could then be transported back to the originating node via the reverse path taken. Such information could be useful during troubleshooting sessions where SNMP is running on nodes but unavailable due to logical layer issues.

7.2   Future Work on XPLANE-based Tools

The tools presented in Chapters 4 and 5 are largely proofs of concept to demonstrate the possibilities with XPLANE-based tools. There is plenty of room for improvement in all

three major components.

7.2.1   Improving the Tactical Edge Suite

The tactical edge suite only consists of four applications at this time. These applications are largely focused on network discovery. The suite is lacking in applications that make better use of packet capture and injection to discover interesting things about the network.

One example of this would be enumerating all logical paths between a source and destination. This could be broken down into an application that attempts to discover all physical paths between a source and destination and an application that tests a given physical path for a corresponding logical path.

Chapter 6 demonstrated the need for applications that detect faulty paths. In the presented case study, the troubleshooting workstation was unable to detect anomalies in Alpha Company's router with the current tactical edge suite. An application to detect faulty reverse paths could be made. In the case of Alpha's router, the application would schedule a packet capture on the interface to Headquarters (eth1) as well as the interface to Alpha's subnet (eth0). Next it would initiate an echo request from Headquarters to a workstation in Alpha's subnet. If an echo reply is seen at eth0 and *not* eth1, there is a faulty reverse path.

Another useful addition to the tactical edge suite would be an application that relocates to routers and enumerates all hosts (including non XPLANE-enabled hosts) on a given interface. This task is currently performed manually by the XPLANE server operator and is unsuitable for enumeration of large subnets.

7.2.2   Improving the XPLANE Server

At this time the XPLANE server can only launch XPLANE applications from the command-line interface. There is no support for launching applications on behalf of a web-based client. Additionally the XPLANE server has been built around the concept of a single user that runs applications in sequential manner. Adding support to take commands from web-based clients will necessitate either an application queueing system or a way to block requests from clients (as well as the command line) while an application is running.

7.2.3 Improving the Web-based UI

The web-based UI has no support for launching XPLANE applications on the XPLANE server. This support would require the UI to be aware of the applications available on the XPLANE server as well as how to specify the necessary parameters to launch the application. Ideally users would specify parameters using the graph by selecting elements in the graph such as nodes and adjacencies.

Anomaly detection is currently performed within the UI. As XPLANE gains capabilities (e.g., the ability to query routing tables) the UI will need to be augmented to discover anomalies in the provided data. Anomalies are currently reported to the user via the anomaly table within the UI. The anomaly table could be improved to provide additional contextual information on each anomaly to ease in troubleshooting. For instance, clicking on an entry in the anomaly table could highlight nodes in the graph that are involved with the detected anomaly.

The UI is also lacking basic navigation features within the graph such as pan and zoom. Such features will be necessary to navigate larger networks using the force-directed graph.

7.3 Conclusions

The problem of troubleshooting a geographically-dispersed network in a combat zone is a hard one. Tactical networks will only get more complex from here and further exacerbate the issue. Units will always plan to have trained people in the right place at the right time, but it rarely works that way. Even after our recent experiences in Iraq and Afghanistan we have no good solution to this problem. The Marine Corps Center for Lessons Learned (MCCLL) web site is host to recent after action reports from units that cite the same communications issues experienced in this thesis. How do we cope with logically broken data systems when no one can reach them? We simply do not have the tools to answer this question. The Marine Corps and other services need new options.

This thesis scratched the surface of that problem and demonstrated there are options available through existing technology. The addition of new protocols and networking technologies (i.e., active networking) to tactical networks opens up new network management possibilities. We can begin to rely on live network measurements instead of broken network

documentation that may or may not reflect reality. We do not have to rely on a Marine's physical presence at the distant end in order to perform the most mundane of tasks. We can query the network in ways not possible before. We can continue to grow these capabilities if we begin to embrace open networking platforms that will allow for rapid adoption of future networking technologies.

The problems discussed in this thesis are not exclusive to the fighting experienced in Iraq and Afghanistan. These issues will exist in future Marine Corps networks. The landing forces of Marine Expeditionary Units will experience these issues when they cannot communicate back to the command element due to a typo in a router. Mobile ad-hoc networks will become unmanageable when the network operations center has no way to gain a live view of the network. The networks of tomorrow will suffer from the problems of today if we do not address our core networking shortfalls.

APPENDIX A:
Code Listings

Listing A.1: Use OnFlood to find a node ID '2' in the XPLANE

```
1  (letrec ((findnode
2     (lambda (n path)
3       (if (equal? n node)
4         (cons node path)
5         (if (not (member node path))
6           (let ((n2 node))
7             (onflood (findnode n (cons n2 path)))))))))
8     (findnode 2 '()))
```

Listing A.2: Send an 'ping' on interface 1 to 10.0.0.1 and look for reply

```
1  (letrecp ((p (lambda (pkts)
2     (if (null? pkts)
3       #f
4       (if (equal? "10.0.0.1" (ip.src (car pkts)))
5         #t
6         (p (cdr pkts))))))))
7     (send 1 '("EchoRequest" "10.0.0.1")
8       (pcap 1 "icmp" 3 p)))
```

Listing A.3: Discover all XPLANE nodes using OnFlood

```
1  (letrec ((f
2     (lambda (path)
3       (if (member node path)
4         path
5         (let ((n node))
6           (onflood (f (cons n path)))))))))
7     (f '()))
```

Listing A.4: Discover all XPLANE nodes using depth-first search

61

```
1  (letrec ((visit
2    (lambda (set)
3      (letrec ((neighbors
4        (lambda (list s1)
5          (if (null? list)
6              s1
7              (neighbors (cdr list) (on (car list) (visit s1)))))))
8        (letrec ((ifaces
9          (lambda (l s2)
10           (if (null? l)
11               s1
12               (ifaces (cdr l) (neighbors (node.direct (car l))
                     s2))))))
13         (if (member node set)
14             set
15             (ifaces (node.ifaces) (cons node set)))))))))
16   (visit '()))
```

Listing A.5: Enumerate information on a remote node

```
1  ; the variable path_to_dest must be defined and be a valid
2  ; path to the destination node ID
3  (letrec ((enuminfo
4    (lambda (lst devs)
5      (if (null? devs)
6          lst
7          (let ((d (car devs)))
8            (enuminfo (append lst (list (list d (node.ip d)
                  (node.mask d) (node.ethaddr d) (node.direct d))))
                  (cdr devs)))))))
9    (letrec ((traverse
10     (lambda (path)
11       (if (and (equal? (car path) node) (null? (cdr path)))
12           (enuminfo (list node.type) (node.ifaces))
13           (let ((newpath (cdr path)))
14             (on (car newpath) (traverse newpath)))))))
```

```
15      (traverse path_to_dest)))
```

Listing A.6: Re-positional Ping

```
1  ; the variables dest_ip, dest_interface, and path_to_dest
2  ; must be defined
3  (letrecp ((proc
4    (lambda (pkts src)
5      (if (null? pkts)
6        #f
7        (if (equal? src (ip.src (car pkts)))
8          #t
9          (proc (cdr pkts) src))))))
10     (letrec ((ping
11       (lambda (dest dev)
12         (send dev (list "EchoRequest" dest)
13            (pcap dev "icmp" 3 (lambda (p) (proc p dest)))))))
14       (letrec ((traverse
15           (lambda (path)
16             (if (and (equal? (car path) node) (null? (cdr
                   path)))
17               (ping dest_ip dest_interface)
18               (let ((newpath (cdr path)))
19                 (on (car newpath) (traverse newpath)))))))
20           (traverse path_to_dest))))
```

THIS PAGE INTENTIONALLY LEFT BLANK

APPENDIX B:
Case Study Network Configuration

Location	Type	Interface - IP/Mask	Routing Table
HQ	Router	eth0 - 10.0.1.1/24 eth1 - 10.0.10.1/30 eth2 - 10.0.10.9/30 eth3 - 10.0.10.5/30	10.0.1.0/24 dev eth0 10.0.2.0/24 via 10.0.10.2 dev eth1 10.0.3.0/24 via 10.0.10.6 dev eth3 10.0.4.0/24 via 10.0.10.10 dev eth2
HQ	Workstation	eth0 - 10.0.1.50/24	default via 10.0.1.1 dev eth0
HQ	Workstation	eth0 - 10.0.1.51/24	default via 10.0.1.1 dev eth0
Alpha	Router	eth0 - 10.0.2.1/24 eth1 - 10.0.10.2/30 eth2 - 10.0.10.13/30	10.0.2.0/24 dev eth0 10.0.1.0/24 via 10.0.10.1 dev eth1[1] 10.0.3.0/24 via 10.0.10.1 dev eth1 10.0.4.0/24 via 10.0.10.14 dev eth2
Alpha	Workstation	eth0 - 10.0.2.50/24	default via 10.0.2.1 dev eth0
Alpha	Workstation	eth0 - 10.0.2.51/24	default via 10.0.2.1 dev eth0
Bravo	Router	eth0 - 10.0.3.1/24 eth1 - 10.0.10.6/31[2]	10.0.3.0/24 dev eth0 10.0.1.0/24 via 10.0.10.5 dev eth1 10.0.2.0/24 via 10.0.10.5 dev eth1 10.0.4.0/24 via 10.0.10.5 dev eth1
Bravo	Workstation	eth0 - 10.0.3.50/24	default via 10.0.3.1 dev eth0
Bravo	Workstation	eth0 - 10.0.3.51/24	default via 10.0.3.1 dev eth0
Charlie	Router	eth0 - 10.0.4.1/24 eth1 - 10.0.10.9/30[3] eth2 - 10.0.10.14/30	10.0.4.0/24 dev eth0 10.0.1.0/24 via 10.0.10.9 dev eth1 10.0.2.0/24 via 10.0.10.13 dev eth2 10.0.3.0/24 via 10.0.10.9 dev eth1
Charlie	Workstation	eth0 - 10.0.4.50/24	default via 10.0.4.1 dev eth0
Charlie	Workstation	eth0 - 10.0.4.51/24	default via 10.0.4.1 dev eth0

Table B.1: Case study network configuration

---

[1]These required routing entries have been accidentally deleted on the router during troubleshooting.
[2]The subnet mask is incorrect. It should be /30.
[3]The IP address is incorrect. The last octet should be 10.

THIS PAGE INTENTIONALLY LEFT BLANK

References

[1] J. Kurose and K. Ross, *Computer Networking: A Top-Down Approach*. Boston, MA: Addison Wesley Higher Education, 2010.

[2] IEEE Standard for Local and Metropolitan Area Networks– Station and Media Access Control Connectivity Discovery, *IEEE Standard 802.1AB-2009*, 2009.

[3] Microsoft. (2010, Sep.). Link layer topology discovery protocol specification. [Online]. Available: http://msdn.microsoft.com/en-us/windows/hardware/gg463061.aspx

[4] Sourceforge. (2014, Jan.). Linkloop for Linux. [Online]. Available: http://sourceforge.net/projects/linkloop/

[5] D. L. Tennenhouse and D. J. Wetherall, "Towards an active network architecture," *Computer Communication Review*, vol. 26, pp. 5–18, 1996.

[6] N. Feamster, J. Rexford, and E. Zegura, "The road to SDN," *Queue*, vol. 11, no. 12, pp. 20–40, Dec. 2013.

[7] M. Clement and D. Volpano, "Programmable diagnostic network measurement with localization and traffic observation," in *Automated Security Management*, E. Al-Shaer, X. Ou, and G. Xie, Eds. Springer International Publishing, 2013, pp. 153–167.

[8] Microsoft. (2002, Feb.). The OSI Model's seven layers defined and functions explained. [Online]. Available: http://support.microsoft.com/kb/103884

[9] U. Hengartner, S. Moon, R. Mortier, and C. Diot, "Detection and analysis of routing loops in packet traces," in *Proceedings of the 2Nd ACM SIGCOMM Workshop on Internet Measurment*, New York, NY, USA, 2002, pp. 107–112.

[10] Cisco. Cisco Discovery Protocol Version 2. [Online]. Available: http://www.cisco.com/en/US/docs/ios-xml/ios/cdp/configuration/15-mt/nm-cdp-discover.pdf

[11] M. Wawrzoniak, L. Peterson, and T. Roscoe, "Sophia: An Information Plane for Networked Systems," *SIGCOMM Comput. Commun. Rev.*, vol. 34, no. 1, pp. 15–20, Jan. 2004.

[12] A. Shieh, E. G. Sirer, and F. B. Schneider, "Netquery: A knowledge plane for reasoning about network properties," *SIGCOMM Comput. Commun. Rev.*, vol. 41, no. 4, pp. 278–289, Aug. 2011.

[13] M. Hicks, P. Kakkar, J. T. Moore, C. A. Gunter, and S. Nettles, "Plan: A programming language for active networks," *ACM SIGPLAN Notices*, vol. 34, pp. 86–93, 1999.

[14] B. Schwartz, A. W. Jackson, W. T. Strayer, W. Zhou, R. D. Rockwell, and C. Partridge, "Smart packets: Applying active networks to network management," *ACM Transactions on Computer Systems*, vol. 18, no. 1, pp. 67–88, Feb. 2000.

[15] J. Mitchell, *Concepts in Programming Languages*. New York, NY: Cambridge University Press, 2007.

[16] Naval Research Laboratory. (2014, Apr.). Common Open Research Emulator (CORE). [Online]. Available: http://www.nrl.navy.mil/itd/ncs/products/core

[17] Eclipse Foundation. (2014, Apr.). The Jetty project. [Online]. Available: http://www.eclipse.org/jetty/about.php

[18] I. Fette and A. Melnikov. (2011, Dec.). The WebSocket Protocol. RFC 6455 (Proposed Standard). Internet Engineering Task Force. [Online]. Available: http://www.ietf.org/rfc/rfc6455.txt

[19] M. Bostock. (2014, Apr.). Data-driven documents. [Online]. Available: http://d3js.org

[20] N. Ferguson and B. Schneier, *Practical Cryptography*. Indianapolis, IN: Wiley Publishing, Inc., 2003.

[21] H. Krawczyk, M. Bellare, and R. Canetti. (1997, Feb.). HMAC: Keyed-Hashing for Message Authentication. RFC 2104 (Proposed Standard). Internet Engineering Task Force. [Online]. Available: http://www.ietf.org/rfc/rfc2104.txt

Initial Distribution List

1. Defense Technical Information Center
   Ft. Belvoir, Virginia

2. Dudley Knox Library
   Naval Postgraduate School
   Monterey, California